Keywords and Punctuation

The following keywords are reserved and can’t be used as identifiers, unless they’re escaped with backticks, as described above in [Identifiers](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID412). Keywords other than inout, var, and let can be used as parameter names in a function declaration or function call without being escaped with backticks. When a member has the same name as a keyword, references to that member don’t need to be escaped with backticks, except when there’s ambiguity between referring to the member and using the keyword—for example, self, Type, and Protocol have special meaning in an explicit member expression, so they must be escaped with backticks in that context.

* Keywords used in declarations: associatedtype, class, deinit, enum, extension, fileprivate, func, import, init, inout, internal, let, open, operator, private, protocol, public, rethrows, static, struct, subscript, typealias, and var.
* Keywords used in statements: break, case, continue, default, defer, do, else, fallthrough, for, guard, if, in, repeat, return, switch, where, and while.
* Keywords used in expressions and types: as, Any, catch, false, is, nil, super, self, Self, throw, throws, true, and try.
* Keywords used in patterns: \_.
* Keywords that begin with a number sign (#): #available, #colorLiteral, #column, #else, #elseif, #endif, #error, #file, #fileLiteral, #function, #if, #imageLiteral, #line, #selector, #sourceLocation, and #warning.
* Keywords reserved in particular contexts: associativity, convenience, dynamic, didSet, final, get, infix, indirect, lazy, left, mutating, none, nonmutating, optional, override, postfix, precedence, prefix, Protocol, required, right, set, Type, unowned, weak, and willSet. Outside the context in which they appear in the grammar, they can be used as identifiers.

The following tokens are reserved as punctuation and can’t be used as custom operators: (, ), {, }, [, ], ., ,, :, ;, =, @, #, & (as a prefix operator), ->, `, ?, and ! (as a postfix operator).

Literals

A *literal* is the source code representation of a value of a type, such as a number or string.

The following are examples of literals:

1. 42 // Integer literal
2. 3.14159 // Floating-point literal
3. "Hello, world!" // String literal
4. true // Boolean literal

A literal doesn’t have a type on its own. Instead, a literal is parsed as having infinite precision and Swift’s type inference attempts to infer a type for the literal. For example, in the declaration let x: Int8 = 42, Swift uses the explicit type annotation (: Int8) to infer that the type of the integer literal 42 is Int8. If there isn’t suitable type information available, Swift infers that the literal’s type is one of the default literal types defined in the Swift standard library. The default types are Int for integer literals, Double for floating-point literals, String for string literals, and Bool for Boolean literals. For example, in the declaration let str = "Hello, world", the default inferred type of the string literal "Hello, world" is String.

When specifying the type annotation for a literal value, the annotation’s type must be a type that can be instantiated from that literal value. That is, the type must conform to one of the following Swift standard library protocols: ExpressibleByIntegerLiteral for integer literals, ExpressibleByFloatLiteral for floating-point literals, ExpressibleByStringLiteral for string literals, ExpressibleByBooleanLiteral for Boolean literals, ExpressibleByUnicodeScalarLiteral for string literals that contain only a single Unicode scalar, and ExpressibleByExtendedGraphemeClusterLiteral for string literals that contain only a single extended grapheme cluster. For example, Int8 conforms to the ExpressibleByIntegerLiteral protocol, and therefore it can be used in the type annotation for the integer literal 42 in the declaration let x: Int8 = 42.

### **Integer Literals**

Integer literals represent integer values of unspecified precision. By default, integer literals are expressed in decimal; you can specify an alternate base using a prefix. Binary literals begin with 0b, octal literals begin with 0o, and hexadecimal literals begin with 0x.

Decimal literals contain the digits 0 through 9. Binary literals contain 0 and 1, octal literals contain 0 through 7, and hexadecimal literals contain 0 through 9 as well as A through F in upper- or lowercase.

Negative integers literals are expressed by prepending a minus sign (-) to an integer literal, as in -42.

Underscores (\_) are allowed between digits for readability, but they’re ignored and therefore don’t affect the value of the literal. Integer literals can begin with leading zeros (0), but they’re likewise ignored and don’t affect the base or value of the literal.

Unless otherwise specified, the default inferred type of an integer literal is the Swift standard library type Int. The Swift standard library also defines types for various sizes of signed and unsigned integers, as described in [Integers](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID317).

### **Floating-Point Literals**

Floating-point literals represent floating-point values of unspecified precision.

By default, floating-point literals are expressed in decimal (with no prefix), but they can also be expressed in hexadecimal (with a 0x prefix).

Decimal floating-point literals consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point (.) followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase e prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the e is multiplied by. For example, 1.25e2 represents 1.25 x 102, which evaluates to 125.0. Similarly, 1.25e-2 represents 1.25 x 10-2, which evaluates to 0.0125.

Hexadecimal floating-point literals consist of a 0x prefix, followed by an optional hexadecimal fraction, followed by a hexadecimal exponent. The hexadecimal fraction consists of a decimal point followed by a sequence of hexadecimal digits. The exponent consists of an upper- or lowercase p prefix followed by a sequence of decimal digits that indicates what power of 2 the value preceding the p is multiplied by. For example, 0xFp2 represents 15 x 22, which evaluates to 60. Similarly, 0xFp-2 represents 15 x 2-2, which evaluates to 3.75.

Negative floating-point literals are expressed by prepending a minus sign (-) to a floating-point literal, as in -42.5.

Underscores (\_) are allowed between digits for readability, but they’re ignored and therefore don’t affect the value of the literal. Floating-point literals can begin with leading zeros (0), but they’re likewise ignored and don’t affect the base or value of the literal.

Unless otherwise specified, the default inferred type of a floating-point literal is the Swift standard library type Double, which represents a 64-bit floating-point number. The Swift standard library also defines a Float type, which represents a 32-bit floating-point number.

### **String Literals**

A string literal is a sequence of characters surrounded by quotation marks. A single-line string literal is surrounded by double quotation marks and has the following form:

1. "characters"

String literals can’t contain an unescaped double quotation mark ("), an unescaped backslash (\), a carriage return, or a line feed.

A multiline string literal is surrounded by three double quotation marks and has the following form:

1. """
2. characters
3. """

Unlike a single-line string literal, a multiline string literal can contain unescaped double quotation marks ("), carriage returns, and line feeds. It can’t contain three unescaped double quotation marks next to each other.

The line break after the """ that begins the multiline string literal is not part of the string. The line break before the """ that ends the literal is also not part of the string. To make a multiline string literal that begins or ends with a line feed, write a blank line as its first or last line.

A multiline string literal can be indented using any combination of spaces and tabs; this indentation is not included in the string. The """ that ends the literal determines the indentation: Every nonblank line in the literal must begin with exactly the same indentation that appears before the closing """; there’s no conversion between tabs and spaces. You can include additional spaces and tabs after that indentation; those spaces and tabs appear in the string.

Line breaks in a multiline string literal are normalized to use the line feed character. Even if your source file has a mix of carriage returns and line feeds, all of the line breaks in the string will be the same.

In a multiline string literal, writing a backslash (\) at the end of a line omits that line break from the string. Any whitespace between the backslash and the line break is also omitted. You can use this syntax to hard wrap a multiline string literal in your source code, without changing the value of the resulting string.

Special characters can be included in string literals of both the single-line and multiline forms using the following escape sequences:

* Null character (\0)
* Backslash (\\)
* Horizontal tab (\t)
* Line feed (\n)
* Carriage return (\r)
* Double quotation mark (\")
* Single quotation mark (\')
* Unicode scalar (\u{n}), where n is a hexadecimal number that has one to eight digits

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (\). The interpolated expression can contain a string literal, but can’t contain an unescaped backslash, a carriage return, or a line feed.

For example, all of the following string literals have the same value:

1. "1 2 3"
2. "1 2 \("3")"
3. "1 2 \(3)"
4. "1 2 \(1 + 2)"
5. let x = 3; "1 2 \(x)"

A string delimited by extended delimiters is a sequence of characters surrounded by quotation marks and a balanced set of one or more number signs (#). A string delimited by extended delimiters has the following forms:

1. #"characters"#
2. #"""
3. characters
4. """#

Special characters in a string delimited by extended delimiters appear in the resulting string as normal characters rather than as special characters. You can use extended delimiters to create strings with characters that would ordinarily have a special effect such as generating a string interpolation, starting an escape sequence, or terminating the string.

The following example shows a string literal and a string delimited by extended delimiters that create equivalent string values:

1. let string = #"\(x) \ " \u{2603}"#
2. let escaped = "\\(x) \\ \" \\u{2603}"
3. print(string)
4. // Prints "\(x) \ " \u{2603}"
5. print(string == escaped)
6. // Prints "true"

If you use more than one number sign to form a string delimited by extended delimiters, don’t place whitespace in between the number signs:

1. print(###"Line 1\###nLine 2"###) // OK
2. print(# # #"Line 1\# # #nLine 2"# # #) // Error

Multiline string literals that you create using extended delimiters have the same indentation requirements as regular multiline string literals.

The default inferred type of a string literal is String. For more information about the String type, see [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) and [String](https://developer.apple.com/documentation/swift/string).

String literals that are concatenated by the + operator are concatenated at compile time. For example, the values of textA and textB in the example below are identical—no runtime concatenation is performed.

1. let textA = "Hello " + "world"
2. let textB = "Hello world"

# **Types**

In Swift, there are two kinds of types: named types and compound types. A named type is a type that can be given a particular name when it’s defined. Named types include classes, structures, enumerations, and protocols. For example, instances of a user-defined class named MyClass have the type MyClass. In addition to user-defined named types, the Swift standard library defines many commonly used named types, including those that represent arrays, dictionaries, and optional values.

Data types that are normally considered basic or primitive in other languages—such as types that represent numbers, characters, and strings—are actually named types, defined and implemented in the Swift standard library using structures. Because they’re named types, you can extend their behavior to suit the needs of your program, using an extension declaration, discussed in [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html) and [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378).

A compound type is a type without a name, defined in the Swift language itself. There are two compound types: function types and tuple types. A compound type may contain named types and other compound types. For example, the tuple type (Int, (Int, Int)) contains two elements: The first is the named type Int, and the second is another compound type (Int, Int).

You can put parentheses around a named type or a compound type. However, adding parentheses around a type doesn’t have any effect. For example, (Int) is equivalent to Int.

## Type Annotation

A type annotation explicitly specifies the type of a variable or expression. Type annotations begin with a colon (:) and end with a type, as the following examples show:

1. let someTuple: (Double, Double) = (3.14159, 2.71828)
2. func someFunction(a: Int) { /\* ... \*/ }

In the first example, the expression someTuple is specified to have the tuple type (Double, Double). In the second example, the parameter a to the function someFunction is specified to have the type Int.

Type annotations can contain an optional list of type attributes before the type

## Metatype Type

A metatype type refers to the type of any type, including class types, structure types, enumeration types, and protocol types.

The metatype of a class, structure, or enumeration type is the name of that type followed by .Type. The metatype of a protocol type—not the concrete type that conforms to the protocol at runtime—is the name of that protocol followed by .Protocol. For example, the metatype of the class type SomeClass is SomeClass.Type and the metatype of the protocol SomeProtocol is SomeProtocol.Protocol.

You can use the postfix self expression to access a type as a value. For example, SomeClass.self returns SomeClass itself, not an instance of SomeClass. And SomeProtocol.self returns SomeProtocol itself, not an instance of a type that conforms to SomeProtocol at runtime. You can call the type(of:) function with an instance of a type to access that instance’s dynamic, runtime type as a value, as the following example shows:

1. class SomeBaseClass {
2. class func printClassName() {
3. print("SomeBaseClass")
4. }
5. }
6. class SomeSubClass: SomeBaseClass {
7. override class func printClassName() {
8. print("SomeSubClass")
9. }
10. }
11. let someInstance: SomeBaseClass = SomeSubClass()
12. // The compile-time type of someInstance is SomeBaseClass,
13. // and the runtime type of someInstance is SomeSubClass
14. type(of: someInstance).printClassName()
15. // Prints "SomeSubClass"

For more information, see [type(of:)](https://developer.apple.com/documentation/swift/2885064-type) in the Swift standard library.

Use an initializer expression to construct an instance of a type from that type’s metatype value. For class instances, the initializer that’s called must be marked with the required keyword or the entire class marked with the final keyword.

1. class AnotherSubClass: SomeBaseClass {
2. let string: String
3. required init(string: String) {
4. self.string = string
5. }
6. override class func printClassName() {
7. print("AnotherSubClass")
8. }
9. }
10. let metatype: AnotherSubClass.Type = AnotherSubClass.self
11. let anotherInstance = metatype.init(string: "some string")

GRAMMAR OF A METATYPE TYPE

*metatype-type* → [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) **.** **Type** | [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) **.** **Protocol**

## Self Type

The Self type isn’t a specific type, but rather lets you conveniently refer to the current type without repeating or knowing that type’s name.

In a protocol declaration or a protocol member declaration, the Self type refers to the eventual type that conforms to the protocol.

In a structure, class, or enumeration declaration, the Self type refers to the type introduced by the declaration. Inside the declaration for a member of a type, the Self type refers to that type. In the members of a class declaration, Self can appear as the return type of a method and in the body of a method, but not in any other context. For example, the code below shows an instance method f whose return type is Self.

1. class Superclass {
2. func f() -> Self { return self }
3. }
4. let x = Superclass()
5. print(type(of: x.f()))
6. // Prints "Superclass"
7. class Subclass: Superclass { }
8. let y = Subclass()
9. print(type(of: y.f()))
10. // Prints "Subclass"
11. let z: Superclass = Subclass()
12. print(type(of: z.f()))
13. // Prints "Subclass"

The last part of the example above shows that Self refers to the runtime type Subclass of the value of z, not the compile-time type Superclass of the variable itself.

Inside a nested type declaration, the Self type refers to the type introduced by the innermost type declaration.

The Self type refers to the same type as the [type(of:)](https://developer.apple.com/documentation/swift/2885064-type) function in the Swift standard library. Writing Self.someStaticMember to access a member of the current type is the same as writing type(of: self).someStaticMember.

### **Try Operator**

A try expression consists of the try operator followed by an expression that can throw an error. It has the following form:

1. try expression

An optional-try expression consists of the try? operator followed by an expression that can throw an error. It has the following form:

1. try? expression

If the expression does not throw an error, the value of the optional-try expression is an optional containing the value of the expression. Otherwise, the value of the optional-try expression is nil.

A forced-try expression consists of the try! operator followed by an expression that can throw an error. It has the following form:

1. try! expression

If the expression throws an error, a runtime error is produced.

When the expression on the left-hand side of a binary operator is marked with try, try?, or try!, that operator applies to the whole binary expression. That said, you can use parentheses to be explicit about the scope of the operator’s application.

1. sum = try someThrowingFunction() + anotherThrowingFunction() // try applies to both function calls
2. sum = try (someThrowingFunction() + anotherThrowingFunction()) // try applies to both function calls
3. sum = (try someThrowingFunction()) + anotherThrowingFunction() // Error: try applies only to the first function call

A try expression can’t appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the try expression is enclosed in parentheses.

### **Literal Expression**

A literal expression consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, a playground literal, or one of the following special literals:

| **Literal** | **Type** | **Value** |
| --- | --- | --- |
| #file | String | The name of the file in which it appears. |
| #line | Int | The line number on which it appears. |
| #column | Int | The column number in which it begins. |
| #function | String | The name of the declaration in which it appears. |
| #dsohandle | UnsafeRawPointer | The DSO (dynamic shared object) handle in use where it appears. |

Inside a function, the value of #function is the name of that function, inside a method it is the name of that method, inside a property getter or setter it is the name of that property, inside special members like init or subscript it is the name of that keyword, and at the top level of a file it is the name of the current module.

When used as the default value of a function or method parameter, the special literal’s value is determined when the default value expression is evaluated at the call site.

1. func logFunctionName(string: String = #function) {
2. print(string)
3. }
4. func myFunction() {
5. logFunctionName() // Prints "myFunction()".
6. }

An array literal is an ordered collection of values. It has the following form:

1. [value 1, value 2, ...]

The last expression in the array can be followed by an optional comma. The value of an array literal has type [T], where T is the type of the expressions inside it. If there are expressions of multiple types, T is their closest common supertype. Empty array literals are written using an empty pair of square brackets and can be used to create an empty array of a specified type.

1. var emptyArray: [Double] = []

A dictionary literal is an unordered collection of key-value pairs. It has the following form:

1. [key 1: value 1, key 2: value 2, ...]

The last expression in the dictionary can be followed by an optional comma. The value of a dictionary literal has type [Key: Value], where Key is the type of its key expressions and Value is the type of its value expressions. If there are expressions of multiple types, Key and Value are the closest common supertype for their respective values. An empty dictionary literal is written as a colon inside a pair of brackets ([:]) to distinguish it from an empty array literal. You can use an empty dictionary literal to create an empty dictionary literal of specified key and value types.

1. var emptyDictionary: [String: Double] = [:]

A playground literal is used by Xcode to create an interactive representation of a color, file, or image within the program editor. Playground literals in plain text outside of Xcode are represented using a special literal syntax.

# **Statements**

In Swift, there are three kinds of statements: simple statements, compiler control statements, and control flow statements. Simple statements are the most common and consist of either an expression or a declaration. Compiler control statements allow the program to change aspects of the compiler’s behavior and include a conditional compilation block and a line control statement.

Control flow statements are used to control the flow of execution in a program. There are several types of control flow statements in Swift, including loop statements, branch statements, and control transfer statements. Loop statements allow a block of code to be executed repeatedly, branch statements allow a certain block of code to be executed only when certain conditions are met, and control transfer statements provide a way to alter the order in which code is executed. In addition, Swift provides a do statement to introduce scope, and catch and handle errors, and a defer statement for running cleanup actions just before the current scope exits.

A semicolon (;) can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

## Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has three loop statements: a for-in statement, a while statement, and a repeat-while statement.

Control flow in a loop statement can be changed by a break statement and a continue statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) and [Continue Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID442) below.

GRAMMAR OF A LOOP STATEMENT

*loop-statement* → [for-in-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_for-in-statement)

*loop-statement* → [while-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_while-statement)

*loop-statement* → [repeat-while-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_repeat-while-statement)

### **For-In Statement**

A for-in statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the [Sequence](https://developer.apple.com/documentation/swift/sequence) protocol.

A for-in statement has the following form:

1. for item in collection {
2. statements
3. }

The makeIterator() method is called on the collection expression to obtain a value of an iterator type—that is, a type that conforms to the [IteratorProtocol](https://developer.apple.com/documentation/swift/iteratorprotocol) protocol. The program begins executing a loop by calling the next() method on the iterator. If the value returned is not nil, it is assigned to the item pattern, the program executes the statements, and then continues execution at the beginning of the loop. Otherwise, the program does not perform assignment or execute the statements, and it is finished executing the for-in statement.

GRAMMAR OF A FOR-IN STATEMENT

*for-in-statement* → **for** **case***opt* [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) **in** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

### **While Statement**

A while statement allows a block of code to be executed repeatedly, as long as a condition remains true.

A while statement has the following form:

1. while condition {
2. statements
3. }

A while statement is executed as follows:

1. The condition is evaluated.

If true, execution continues to step 2. If false, the program is finished executing the while statement.

1. The program executes the statements, and execution returns to step 1.

Because the value of the condition is evaluated before the statements are executed, the statements in a while statement can be executed zero or more times.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF A WHILE STATEMENT

*while-statement* → **while** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

*condition-list* → [condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition) | [condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition) **,** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list)

*condition* → [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) | [availability-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_availability-condition) | [case-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-condition) | [optional-binding-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_optional-binding-condition)

*case-condition* → **case** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer)

*optional-binding-condition* → **let** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer) | **var** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer)

### **Repeat-While Statement**

A repeat-while statement allows a block of code to be executed one or more times, as long as a condition remains true.

A repeat-while statement has the following form:

1. repeat {
2. statements
3. } while condition

A repeat-while statement is executed as follows:

1. The program executes the statements, and execution continues to step 2.
2. The condition is evaluated.

If true, execution returns to step 1. If false, the program is finished executing the repeat-while statement.

Because the value of the condition is evaluated after the statements are executed, the statements in a repeat-while statement are executed at least once.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF A REPEAT-WHILE STATEMENT

*repeat-while-statement* → **repeat** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) **while** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

## Branch Statements

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements: an if statement, a guard statement, and a switch statement.

Control flow in an if statement or a switch statement can be changed by a break statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) below.

GRAMMAR OF A BRANCH STATEMENT

*branch-statement* → [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

*branch-statement* → [guard-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_guard-statement)

*branch-statement* → [switch-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-statement)

### **If Statement**

An if statement is used for executing code based on the evaluation of one or more conditions.

There are two basic forms of an if statement. In each form, the opening and closing braces are required.

The first form allows code to be executed only when a condition is true and has the following form:

1. if condition {
2. statements
3. }

The second form of an if statement provides an additional else clause (introduced by the else keyword) and is used for executing one part of code when the condition is true and another part of code when the same condition is false. When a single else clause is present, an if statement has the following form:

1. if condition {
2. statements to execute if condition is true
3. } else {
4. statements to execute if condition is false
5. }

The else clause of an if statement can contain another if statement to test more than one condition. An if statement chained together in this way has the following form:

1. if condition 1 {
2. statements to execute if condition 1 is true
3. } else if condition 2 {
4. statements to execute if condition 2 is true
5. } else {
6. statements to execute if both conditions are false
7. }

The value of any condition in an if statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF AN IF STATEMENT

*if-statement* → **if** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) [else-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-clause) *opt*

*else-clause* → **else** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) | **else** [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

### **Guard Statement**

A guard statement is used to transfer program control out of a scope if one or more conditions aren’t met.

A guard statement has the following form:

1. guard condition else {
2. statements
3. }

The value of any condition in a guard statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

Any constants or variables assigned a value from an optional binding declaration in a guard statement condition can be used for the rest of the guard statement’s enclosing scope.

The else clause of a guard statement is required, and must either call a function with the Never return type or transfer program control outside the guard statement’s enclosing scope using one of the following statements:

* return
* break
* continue
* throw

Control transfer statements are discussed in [Control Transfer Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID440) below. For more information on functions with the Never return type, see [Functions that Never Return](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID551).

GRAMMAR OF A GUARD STATEMENT

*guard-statement* → **guard** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) **else** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

### **Switch Statement**

A switch statement allows certain blocks of code to be executed depending on the value of a control expression.

A switch statement has the following form:

1. switch control expression {
2. case pattern 1:
3. statements
4. case pattern 2 where condition:
5. statements
6. case pattern 3 where condition,
7. pattern 4 where condition:
8. statements
9. default:
10. statements
11. }

The control expression of the switch statement is evaluated and then compared with the patterns specified in each case. If a match is found, the program executes the statements listed within the scope of that case. The scope of each case can’t be empty. As a result, you must include at least one statement following the colon (:) of each case label. Use a single break statement if you don’t intend to execute any code in the body of a matched case.

The values of expressions your code can branch on are very flexible. For example, in addition to the values of scalar types, such as integers and characters, your code can branch on the values of any type, including floating-point numbers, strings, tuples, instances of custom classes, and optionals. The value of the control expression can even be matched to the value of a case in an enumeration and checked for inclusion in a specified range of values. For examples of how to use these various types of values in switch statements, see [Switch](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID129) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

A switch case can optionally contain a where clause after each pattern. A where clause is introduced by the where keyword followed by an expression, and is used to provide an additional condition before a pattern in a case is considered matched to the control expression. If a where clause is present, the statements within the relevant case are executed only if the value of the control expression matches one of the patterns of the case and the expression of the where clause evaluates to true. For example, a control expression matches the case in the example below only if it is a tuple that contains two elements of the same value, such as (1, 1).

1. case let (x, y) where x == y:

As the above example shows, patterns in a case can also bind constants using the let keyword (they can also bind variables using the var keyword). These constants (or variables) can then be referenced in a corresponding where clause and throughout the rest of the code within the scope of the case. If the case contains multiple patterns that match the control expression, all of the patterns must contain the same constant or variable bindings, and each bound variable or constant must have the same type in all of the case’s patterns.

A switch statement can also include a default case, introduced by the default keyword. The code within a default case is executed only if no other cases match the control expression. A switch statement can include only one default case, which must appear at the end of the switch statement.

Although the actual execution order of pattern-matching operations, and in particular the evaluation order of patterns in cases, is unspecified, pattern matching in a switch statement behaves as if the evaluation is performed in source order—that is, the order in which they appear in source code. As a result, if multiple cases contain patterns that evaluate to the same value, and thus can match the value of the control expression, the program executes only the code within the first matching case in source order.

#### **Switch Statements Must Be Exhaustive**

In Swift, every possible value of the control expression’s type must match the value of at least one pattern of a case. When this simply isn’t feasible (for example, when the control expression’s type is Int), you can include a default case to satisfy the requirement.

#### **Switching Over Future Enumeration Cases**

A nonfrozen enumeration is a special kind of enumeration that may gain new enumeration cases in the future—even after you compile and ship an app. Switching over a nonfrozen enumeration requires extra consideration. When a library’s authors mark an enumeration as nonfrozen, they reserve the right to add new enumeration cases, and any code that interacts with that enumeration must be able to handle those future cases without being recompiled. Code that’s compiled in library evolution mode, code in the standard library, Swift overlays for Apple frameworks, and C and Objective-C code can declare nonfrozen enumerations. For information about frozen and nonfrozen enumerations, see [frozen](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID620).

When switching over a nonfrozen enumeration value, you always need to include a default case, even if every case of the enumeration already has a corresponding switch case. You can apply the @unknown attribute to the default case, which indicates that the default case should match only enumeration cases that are added in the future. Swift produces a warning if the default case matches any enumeration case that is known at compiler time. This future warning informs you that the library author added a new case to the enumeration that doesn’t have a corresponding switch case.

The following example switches over all three existing cases of the standard library’s [Mirror.AncestorRepresentation](https://developer.apple.com/documentation/swift/mirror/ancestorrepresentation) enumeration. If you add additional cases in the future, the compiler generates a warning to indicate that you need to update the switch statement to take the new cases into account.

1. let representation: Mirror.AncestorRepresentation = .generated
2. switch representation {
3. case .customized:
4. print("Use the nearest ancestor’s implementation.")
5. case .generated:
6. print("Generate a default mirror for all ancestor classes.")
7. case .suppressed:
8. print("Suppress the representation of all ancestor classes.")
9. @unknown default:
10. print("Use a representation that was unknown when this code was compiled.")
11. }
12. // Prints "Generate a default mirror for all ancestor classes."

#### **Execution Does Not Fall Through Cases Implicitly**

After the code within a matched case has finished executing, the program exits from the switch statement. Program execution does not continue or “fall through” to the next case or default case. That said, if you want execution to continue from one case to the next, explicitly include a fallthrough statement, which simply consists of the fallthrough keyword, in the case from which you want execution to continue. For more information about the fallthrough statement, see [Fallthrough Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html" \l "ID443) below.

GRAMMAR OF A SWITCH STATEMENT

*switch-statement* → **switch** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) **{** [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt* **}**

*switch-cases* → [switch-case](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-case) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-case* → [case-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-label) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements)

*switch-case* → [default-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_default-label) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements)

*switch-case* → [conditional-switch-case](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_conditional-switch-case)

*case-label* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **case** [case-item-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-item-list) **:**

*case-item-list* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* | [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* **,** [case-item-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-item-list)

*default-label* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **default** **:**

*where-clause* → **where** [where-expression](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-expression)

*where-expression* → [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

*conditional-switch-case* → [switch-if-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-if-directive-clause) [switch-elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-elseif-directive-clauses) *opt* [switch-else-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-else-directive-clause) *opt* [endif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_endif-directive)

*switch-if-directive-clause* → [if-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-elseif-directive-clauses* → [elseif-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clause) [switch-elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-elseif-directive-clauses) *opt*

*switch-elseif-directive-clause* → [elseif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-else-directive-clause* → [else-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

## Labeled Statement

You can prefix a loop statement, an if statement, a switch statement, or a do statement with a statement label, which consists of the name of the label followed immediately by a colon (:). Use statement labels with break and continue statements to be explicit about how you want to change control flow in a loop statement or a switch statement, as discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) and [Continue Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID442) below.

The scope of a labeled statement is the entire statement following the statement label. You can nest labeled statements, but the name of each statement label must be unique.

For more information and to see examples of how to use statement labels, see [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A LABELED STATEMENT

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [loop-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_loop-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [switch-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [do-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_do-statement)

*statement-label* → [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) **:**

*label-name* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

## Control Transfer Statements

Control transfer statements can change the order in which code in your program is executed by unconditionally transferring program control from one piece of code to another. Swift has five control transfer statements: a break statement, a continue statement, a fallthrough statement, a return statement, and a throw statement.

GRAMMAR OF A CONTROL TRANSFER STATEMENT

*control-transfer-statement* → [break-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_break-statement)

*control-transfer-statement* → [continue-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_continue-statement)

*control-transfer-statement* → [fallthrough-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html" \l "grammar_fallthrough-statement)

*control-transfer-statement* → [return-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_return-statement)

*control-transfer-statement* → [throw-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_throw-statement)

### **Break Statement**

A break statement ends program execution of a loop, an if statement, or a switch statement. A break statement can consist of only the break keyword, or it can consist of the break keyword followed by the name of a statement label, as shown below.

1. break
2. break label name

When a break statement is followed by the name of a statement label, it ends program execution of the loop, if statement, or switch statement named by that label.

When a break statement is not followed by the name of a statement label, it ends program execution of the switch statement or the innermost enclosing loop statement in which it occurs. You can’t use an unlabeled break statement to break out of an if statement.

In both cases, program control is then transferred to the first line of code following the enclosing loop or switch statement, if any.

For examples of how to use a break statement, see [Break](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID137) and [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A BREAK STATEMENT

*break-statement* → **break** [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) *opt*

### **Continue Statement**

A continue statement ends program execution of the current iteration of a loop statement but does not stop execution of the loop statement. A continue statement can consist of only the continue keyword, or it can consist of the continue keyword followed by the name of a statement label, as shown below.

1. continue
2. continue label name

When a continue statement is followed by the name of a statement label, it ends program execution of the current iteration of the loop statement named by that label.

When a continue statement is not followed by the name of a statement label, it ends program execution of the current iteration of the innermost enclosing loop statement in which it occurs.

In both cases, program control is then transferred to the condition of the enclosing loop statement.

In a for statement, the increment expression is still evaluated after the continue statement is executed, because the increment expression is evaluated after the execution of the loop’s body.

For examples of how to use a continue statement, see [Continue](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID136) and [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A CONTINUE STATEMENT

*continue-statement* → **continue** [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) *opt*

### **Fallthrough Statement**

A fallthrough statement consists of the fallthrough keyword and occurs only in a case block of a switch statement. A fallthrough statement causes program execution to continue from one case in a switch statement to the next case. Program execution continues to the next case even if the patterns of the case label do not match the value of the switch statement’s control expression.

A fallthrough statement can appear anywhere inside a switch statement, not just as the last statement of a case block, but it can’t be used in the final case block. It also cannot transfer control into a case block whose pattern contains value binding patterns.

For an example of how to use a fallthrough statement in a switch statement, see [Control Transfer Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID135) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A FALLTHROUGH STATEMENT

*fallthrough-statement* → **fallthrough**

### **Return Statement**

A return statement occurs in the body of a function or method definition and causes program execution to return to the calling function or method. Program execution continues at the point immediately following the function or method call.

A return statement can consist of only the return keyword, or it can consist of the return keyword followed by an expression, as shown below.

1. return
2. return expression

When a return statement is followed by an expression, the value of the expression is returned to the calling function or method. If the value of the expression does not match the value of the return type declared in the function or method declaration, the expression’s value is converted to the return type before it is returned to the calling function or method.

NOTE

As described in [Failable Initializers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html" \l "ID376), a special form of the return statement (return nil) can be used in a failable initializer to indicate initialization failure.

When a return statement is not followed by an expression, it can be used only to return from a function or method that does not return a value (that is, when the return type of the function or method is Void or ()).

GRAMMAR OF A RETURN STATEMENT

*return-statement* → **return** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) *opt*

### **Throw Statement**

A throw statement occurs in the body of a throwing function or method, or in the body of a closure expression whose type is marked with the throws keyword.

A throw statement causes a program to end execution of the current scope and begin error propagation to its enclosing scope. The error that’s thrown continues to propagate until it’s handled by a catch clause of a do statement.

A throw statement consists of the throw keyword followed by an expression, as shown below.

1. throw expression

The value of the expression must have a type that conforms to the Error protocol.

For an example of how to use a throw statement, see [Propagating Errors Using Throwing Functions](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID510) in [Error Handling](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html).

GRAMMAR OF A THROW STATEMENT

*throw-statement* → **throw** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

## Defer Statement

A defer statement is used for executing code just before transferring program control outside of the scope that the defer statement appears in.

A defer statement has the following form:

1. defer {
2. statements
3. }

The statements within the defer statement are executed no matter how program control is transferred. This means that a defer statement can be used, for example, to perform manual resource management such as closing file descriptors, and to perform actions that need to happen even if an error is thrown.

If multiple defer statements appear in the same scope, the order they appear is the reverse of the order they are executed. Executing the last defer statement in a given scope first means that statements inside that last defer statement can refer to resources that will be cleaned up by other defer statements.

1. func f() {
2. defer { print("First defer") }
3. defer { print("Second defer") }
4. print("End of function")
5. }
6. f()
7. // Prints "End of function"
8. // Prints "Second defer"
9. // Prints "First defer"

The statements in the defer statement can’t transfer program control outside of the defer statement.

GRAMMAR OF A DEFER STATEMENT

*defer-statement* → **defer** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

## Do Statement

The do statement is used to introduce a new scope and can optionally contain one or more catch clauses, which contain patterns that match against defined error conditions. Variables and constants declared in the scope of a do statement can be accessed only within that scope.

A do statement in Swift is similar to curly braces ({}) in C used to delimit a code block, and does not incur a performance cost at runtime.

A do statement has the following form:

1. do {
2. try expression
3. statements
4. } catch pattern 1 {
5. statements
6. } catch pattern 2 where condition {
7. statements
8. }

Like a switch statement, the compiler attempts to infer whether catch clauses are exhaustive. If such a determination can be made, the error is considered handled. Otherwise, the error can propagate out of the containing scope, which means the error must be handled by an enclosing catch clause or the containing function must be declared with throws.

To ensure that an error is handled, use a catch clause with a pattern that matches all errors, such as a wildcard pattern (\_). If a catch clause does not specify a pattern, the catch clause matches and binds any error to a local constant named error. For more information about the patterns you can use in a catch clause, see [Patterns](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html).

To see an example of how to use a do statement with several catch clauses, see [Handling Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID512).

GRAMMAR OF A DO STATEMENT

*do-statement* → **do** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) [catch-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clauses) *opt*

*catch-clauses* → [catch-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clause) [catch-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clauses) *opt*

*catch-clause* → **catch** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) *opt* [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

## Compiler Control Statements

Compiler control statements allow the program to change aspects of the compiler’s behavior. Swift has three compiler control statements: a conditional compilation block a line control statement, and a compile-time diagnostic statement.

GRAMMAR OF A COMPILER CONTROL STATEMENT

*compiler-control-statement* → [conditional-compilation-block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_conditional-compilation-block)

*compiler-control-statement* → [line-control-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_line-control-statement)

*compiler-control-statement* → [diagnostic-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-statement)

### **Conditional Compilation Block**

A conditional compilation block allows code to be conditionally compiled depending on the value of one or more compilation conditions.

Every conditional compilation block begins with the #if compilation directive and ends with the #endif compilation directive. A simple conditional compilation block has the following form:

1. #if compilation condition
2. statements
3. #endif

Unlike the condition of an if statement, the compilation condition is evaluated at compile time. As a result, the statements are compiled and executed only if the compilation condition evaluates to true at compile time.

The compilation condition can include the true and false Boolean literals, an identifier used with the -D command line flag, or any of the platform conditions listed in the table below.

| **Platform condition** | **Valid arguments** |
| --- | --- |
| os() | macOS, iOS, watchOS, tvOS, Linux |
| arch() | i386, x86\_64, arm, arm64 |
| swift() | >= or < followed by a version number |
| compiler() | >= or < followed by a version number |
| canImport() | A module name |
| targetEnvironment() | simulator, macCatalyst |

The version number for the swift() and compiler() platform conditions consists of a major number, optional minor number, optional patch number, and so on, with a dot (.) separating each part of the version number. There must not be whitespace between the comparison operator and the version number. The version for compiler() is the compiler version, regardless of the Swift version setting passed to the compiler. The version for swift() is the language version currently being compiled. For example, if you compile your code using the Swift 5 compiler in Swift 4.2 mode, the compiler version is 5 and the language version is 4.2. With those settings, the following code prints all three messages:

1. #if compiler(>=5)
2. print("Compiled with the Swift 5 compiler or later")
3. #endif
4. #if swift(>=4.2)
5. print("Compiled in Swift 4.2 mode or later")
6. #endif
7. #if compiler(>=5) && swift(<5)
8. print("Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5")
9. #endif
10. // Prints "Compiled with the Swift 5 compiler or later"
11. // Prints "Compiled in Swift 4.2 mode or later"
12. // Prints "Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5"

The argument for the canImport() platform condition is the name of a module that may not be present on all platforms. This condition tests whether it’s possible to import the module, but doesn’t actually import it. If the module is present, the platform condition returns true; otherwise, it returns false.

The targetEnvironment() platform condition returns true when code is compiled for a simulator; otherwise, it returns false.

NOTE

The arch(arm) platform condition does not return true for ARM 64 devices. The arch(i386) platform condition returns true when code is compiled for the 32–bit iOS simulator.

You can combine compilation conditions using the logical operators &&, ||, and ! and use parentheses for grouping. These operators have the same associativity and precedence as the logical operators that are used to combine ordinary Boolean expressions.

Similar to an if statement, you can add multiple conditional branches to test for different compilation conditions. You can add any number of additional branches using #elseif clauses. You can also add a final additional branch using an #else clause. Conditional compilation blocks that contain multiple branches have the following form:

1. #if compilation condition 1
2. statements to compile if compilation condition 1 is true
3. #elseif compilation condition 2
4. statements to compile if compilation condition 2 is true
5. #else
6. statements to compile if both compilation conditions are false
7. #endif

NOTE

Each statement in the body of a conditional compilation block is parsed even if it’s not compiled. However, there is an exception if the compilation condition includes a swift() platform condition: The statements are parsed only if the compiler’s version of Swift matches what is specified in the platform condition. This exception ensures that an older compiler doesn’t attempt to parse syntax introduced in a newer version of Swift.

GRAMMAR OF A CONDITIONAL COMPILATION BLOCK

*conditional-compilation-block* → [if-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive-clause) [elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clauses) *opt* [else-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive-clause) *opt* [endif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_endif-directive)

*if-directive-clause* → [if-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*elseif-directive-clauses* → [elseif-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clause) [elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clauses) *opt*

*elseif-directive-clause* → [elseif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*else-directive-clause* → [else-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*if-directive* → **#if**

*elseif-directive* → **#elseif**

*else-directive* → **#else**

*endif-directive* → **#endif**

*compilation-condition* → [platform-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_platform-condition)

*compilation-condition* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

*compilation-condition* → [boolean-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html" \l "grammar_boolean-literal)

*compilation-condition* → **(** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **)**

*compilation-condition* → **!** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*compilation-condition* → [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **&&** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*compilation-condition* → [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **||** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*platform-condition* → **os** **(** [operating-system](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_operating-system) **)**

*platform-condition* → **arch** **(** [architecture](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_architecture) **)**

*platform-condition* → **swift** **(** **>=** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)** | **swift** **(** **<** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)**

*platform-condition* → **compiler** **(** **>=** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)** | **compiler** **(** **<** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)**

*platform-condition* → **canImport** **(** [module-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_module-name) **)**

*platform-condition* → **targetEnvironment** **(** [environment](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_environment) **)**

*operating-system* → **macOS** | **iOS** | **watchOS** | **tvOS**

*architecture* → **i386** | **x86\_64** | **arm** | **arm64**

*swift-version* → [decimal-digits](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_decimal-digits) [swift-version-continuation](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version-continuation) *opt*

*swift-version-continuation* → **.** [decimal-digits](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_decimal-digits) [swift-version-continuation](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version-continuation) *opt*

*module-name* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

*environment* → **simulator**

### **Line Control Statement**

A line control statement is used to specify a line number and filename that can be different from the line number and filename of the source code being compiled. Use a line control statement to change the source code location used by Swift for diagnostic and debugging purposes.

A line control statement has the following forms:

1. #sourceLocation(file: filename, line: line number)
2. #sourceLocation()

The first form of a line control statement changes the values of the #line and #file literal expressions, beginning with the line of code following the line control statement. The line number changes the value of #line and is any integer literal greater than zero. The filename changes the value of #file and is a string literal.

The second form of a line control statement, #sourceLocation(), resets the source code location back to the default line numbering and filename.

GRAMMAR OF A LINE CONTROL STATEMENT

*line-control-statement* → **#sourceLocation** **(** **file:** [file-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_file-name) **,** **line:** [line-number](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_line-number) **)**

*line-control-statement* → **#sourceLocation** **(** **)**

*line-number* → A decimal integer greater than zero

*file-name* → [static-string-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_static-string-literal)

### **Compile-Time Diagnostic Statement**

A compile-time diagnostic statement causes the compiler to emit an error or a warning during compilation. A compile-time diagnostic statement has the following forms:

1. #error("error message")
2. #warning("warning message")

The first form emits the error message as a fatal error and terminates the compilation process. The second form emits the warning message as a nonfatal warning and allows compilation to proceed. You write the diagnostic message as a static string literal. Static string literals can’t use features like string interpolation or concatenation, but they can use the multiline string literal syntax.

GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT

*diagnostic-statement* → **#error** **(** [diagnostic-message](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-message) **)**

*diagnostic-statement* → **#warning** **(** [diagnostic-message](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-message) **)**

*diagnostic-message* → [static-string-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_static-string-literal)

## Availability Condition

An availability condition is used as a condition of an if, while, and guard statement to query the availability of APIs at runtime, based on specified platforms arguments.

An availability condition has the following form:

1. if #available(platform name version, ..., \*) {
2. statements to execute if the APIs are available
3. } else {
4. fallback statements to execute if the APIs are unavailable
5. }

You use an availability condition to execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

The availability condition takes a comma-separated list of platform names and versions. Use iOS, macOS, watchOS, and tvOS for the platform names, and include the corresponding version numbers. The \* argument is required and specifies that on any other platform, the body of the code block guarded by the availability condition executes on the minimum deployment target specified by your target.

### **Throwing Functions and Methods**

Functions and methods that can throw an error must be marked with the throws keyword. These functions and methods are known as throwing functions and throwing methods. They have the following form:

1. func function name(parameters) throws -> return type {
2. statements
3. }

Calls to a throwing function or method must be wrapped in a try or try! expression (that is, in the scope of a try or try! operator).

The throws keyword is part of a function’s type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one.

You can’t overload a function based only on whether the function can throw an error. That said, you can overload a function based on whether a function parameter can throw an error.

A throwing method can’t override a nonthrowing method, and a throwing method can’t satisfy a protocol requirement for a nonthrowing method. That said, a nonthrowing method can override a throwing method, and a nonthrowing method can satisfy a protocol requirement for a throwing method.

### **Rethrowing Functions and Methods**

A function or method can be declared with the rethrows keyword to indicate that it throws an error only if one of its function parameters throws an error. These functions and methods are known as rethrowing functions and rethrowing methods. Rethrowing functions and methods must have at least one throwing function parameter.

1. func someFunction(callback: () throws -> Void) rethrows {
2. try callback()
3. }

A rethrowing function or method can contain a throw statement only inside a catch clause. This lets you call the throwing function inside a do-catch block and handle errors in the catch clause by throwing a different error. In addition, the catch clause must handle only errors thrown by one of the rethrowing function’s throwing parameters. For example, the following is invalid because the catch clause would handle the error thrown by alwaysThrows().

1. func alwaysThrows() throws {
2. throw SomeError.error
3. }
4. func someFunction(callback: () throws -> Void) rethrows {
5. do {
6. try callback()
7. try alwaysThrows() // Invalid, alwaysThrows() isn't a throwing parameter
8. } catch {
9. throw AnotherError.error
10. }
11. }

A throwing method can’t override a rethrowing method, and a throwing method can’t satisfy a protocol requirement for a rethrowing method. That said, a rethrowing method can override a throwing method, and a rethrowing method can satisfy a protocol requirement for a throwing method.

# **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:

1. @attribute name
2. @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:
  1. 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:
  1. 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:
  1. 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:
  1. 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:
  1. 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
  2. protocol MyProtocol {
  3. // protocol definition
  4. }
  5. // Subsequent release renames MyProtocol
  6. protocol MyRenamedProtocol {
  7. // protocol definition
  8. }
  9. @available(\*, unavailable, renamed: "MyRenamedProtocol")
  10. 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:

1. @available(platform name version number, \*)
2. @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.

1. @available(iOS 10.0, macOS 10.12, \*)
2. class MyClass {
3. // class definition
4. }

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.

1. @available(swift 3.0.2)
2. @available(macOS 10.12, \*)
3. struct MyStruct {
4. // struct definition
5. }

### **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.

1. @dynamicCallable
2. struct TelephoneExchange {
3. func dynamicallyCall(withArguments phoneNumber: [Int]) {
4. if phoneNumber == [4, 1, 1] {
5. print("Get Swift help on forums.swift.org")
6. } else {
7. print("Unrecognized number")
8. }
9. }
10. }
11. let dial = TelephoneExchange()
12. // Use a dynamic method call.
13. dial(4, 1, 1)
14. // Prints "Get Swift help on forums.swift.org"
15. dial(8, 6, 7, 5, 3, 0, 9)
16. // Prints "Unrecognized number"
17. // Call the underlying method directly.
18. dial.dynamicallyCall(withArguments: [4, 1, 1])

The declaration of the dynamicallyCall(withArguments:) method must have a single parameter that conforms to the [ExpressibleByArrayLiteral](https://developer.apple.com/documentation/swift/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.

1. @dynamicCallable
2. struct Repeater {
3. func dynamicallyCall(withKeywordArguments pairs: KeyValuePairs<String, Int>) -> String {
4. return pairs
5. .map { label, count in
6. repeatElement(label, count: count).joined(separator: " ")
7. }
8. .joined(separator: "\n")
9. }
10. }
11. let repeatLabels = Repeater()
12. print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
13. // a
14. // b b
15. // c c c
16. // b b
17. // a

The declaration of the dynamicallyCall(withKeywordArguments:) method must have a single parameter that conforms to the [ExpressibleByDictionaryLiteral](https://developer.apple.com/documentation/swift/expressiblebydictionaryliteral) protocol, and the return type can be any type. The parameter’s [Key](https://developer.apple.com/documentation/swift/expressiblebydictionaryliteral/2294108-key) must be [ExpressibleByStringLiteral](https://developer.apple.com/documentation/swift/expressiblebystringliteral). The previous example uses [KeyValuePairs](https://developer.apple.com/documentation/swift/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>.

1. 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](https://developer.apple.com/documentation/swift/keypath), [WritableKeyPath](https://developer.apple.com/documentation/swift/writablekeypath), or [ReferenceWritableKeyPath](https://developer.apple.com/documentation/swift/referencewritablekeypath). It can accept member names using an argument of a type that conforms to the [ExpressibleByStringLiteral](https://developer.apple.com/documentation/swift/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:

1. @dynamicMemberLookup
2. struct DynamicStruct {
3. let dictionary = ["someDynamicMember": 325,
4. "someOtherMember": 787]
5. subscript(dynamicMember member: String) -> Int {
6. return dictionary[member] ?? 1054
7. }
8. }
9. let s = DynamicStruct()
10. // Use dynamic member lookup.
11. let dynamic = s.someDynamicMember
12. print(dynamic)
13. // Prints "325"
14. // Call the underlying subscript directly.
15. let equivalent = s[dynamicMember: "someDynamicMember"]
16. print(dynamic == equivalent)
17. // 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:

1. struct Point { var x, y: Int }
2. @dynamicMemberLookup
3. struct PassthroughWrapper<Value> {
4. var value: Value
5. subscript<T>(dynamicMember member: KeyPath<Value, T>) -> T {
6. get { return value[keyPath: member] }
7. }
8. }
9. let point = Point(x: 381, y: 431)
10. let wrapper = PassthroughWrapper(value: point)
11. print(wrapper.x)

### **frozen**

Apply this attribute to a structure or enumeration declaration to restrict the kinds of changes you can make to the type. This attribute is allowed only when compiling in library evolution mode. Future versions of the library can’t change the declaration by adding, removing, or reordering an enumeration’s cases or a structure’s stored instance properties. These changes are allowed on nonfrozen types, but they break ABI compatibility for frozen types.

NOTE

When the compiler isn’t in library evolution mode, all structures and enumerations are implicitly frozen, and you can’t use this attribute.

In library evolution mode, code that interacts with members of nonfrozen structures and enumerations is compiled in a way that allows it to continue working without recompiling even if a future version of the library adds, removes, or reorders some of that type’s members. The compiler makes this possible using techniques like looking up information at runtime and adding a layer of indirection. Marking a structure or enumeration as frozen gives up this flexibility to gain performance: Future versions of the library can make only limited changes to the type, but the compiler can make additional optimizations in code that interacts with the type’s members.

Frozen types, the types of the stored properties of frozen structures, and the associated values of frozen enumeration cases must be public or marked with the usableFromInline attribute. The properties of a frozen structure can’t have property observers, and expressions that provide the initial value for stored instance properties must follow the same restrictions as inlinable functions, as discussed in [inlinable](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID587).

To enable library evolution mode on the command line, pass the -enable-library-evolution option to the Swift compiler. To enable it in Xcode, set the “Build Libraries for Distribution” build setting (BUILD\_LIBRARY\_FOR\_DISTRIBUTION) to Yes, as described in [Xcode Help](https://help.apple.com/xcode/mac/current/" \l "/dev04b3a04ba).

A switch statement over a frozen enumeration doesn’t require a default case, as discussed in [Switching Over Future Enumeration Cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602). Including a default or @unknown default case when switching over a frozen enumeration produces a warning because that code is never executed.

### **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:

1. import AppKit
2. 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](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html#//apple_ref/doc/uid/TP40011210-CH10-SW1) in [Programming with Objective-C](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011210). 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.

1. class ExampleClass: NSObject {
2. @objc var enabled: Bool {
3. @objc(isEnabled) get {
4. // Return the appropriate value
5. }
6. }
7. }

### **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.

### **propertyWrapper**

Apply this attribute to a class, structure, or enumeration declaration to use that type as a property wrapper. When you apply this attribute to a type, you create a custom attribute with the same name as the type. Apply that new attribute to a property of a class, structure, or enumeration to wrap access to the property through an instance of the wrapper type. Local and global variables can’t use property wrappers.

The wrapper must define a wrappedValue instance property. The wrapped value of the property is the value that the getter and setter for this property expose. In most cases, wrappedValue is a computed value, but it can be a stored value instead. The wrapper is responsible for defining and managing any underlying storage needed by its wrapped value. The compiler synthesizes storage for the instance of the wrapper type by prefixing the name of the wrapped property with an underscore (\_)—for example, the wrapper for someProperty is stored as \_someProperty. The synthesized storage for the wrapper has an access control level of private.

A property that has a property wrapper can include willSet and didSet blocks, but it can’t override the compiler-synthesized get or set blocks.

Swift provides two forms of syntactic sugar for initialization of a property wrapper. You can use assignment syntax in the definition of a wrapped value to pass the expression on the right-hand side of the assignment as the argument to the wrappedValue parameter of the property wrapper’s initializer. You can also provide arguments to the attribute when you apply it to a property, and those arguments are passed to the property wrapper’s initializer. For example, in the code below, SomeStruct calls each of the initializers that SomeWrapper defines.

1. @propertyWrapper
2. struct SomeWrapper {
3. var wrappedValue: Int
4. var someValue: Double
5. init() {
6. self.wrappedValue = 100
7. self.someValue = 12.3
8. }
9. init(wrappedValue: Int) {
10. self.wrappedValue = wrappedValue
11. self.someValue = 45.6
12. }
13. init(wrappedValue value: Int, custom: Double) {
14. self.wrappedValue = value
15. self.someValue = custom
16. }
17. }
18. struct SomeStruct {
19. // Uses init()
20. @SomeWrapper var a: Int
21. // Uses init(wrappedValue:)
22. @SomeWrapper var b = 10
23. // Both use init(wrappedValue:custom:)
24. @SomeWrapper(custom: 98.7) var c = 30
25. @SomeWrapper(wrappedValue: 30, custom: 98.7) var d
26. }

The projected value for a wrapped property is a second value that a property wrapper can use to expose additional functionality. The author of a property wrapper type is responsible for determining the meaning of its projected value and defining the interface that the projected value exposes. To project a value from a property wrapper, define a projectedValue instance property on the wrapper type. The compiler synthesizes an identifier for the projected value by prefixing the name of the wrapped property with a dollar sign ($)—for example, the projected value for someProperty is $someProperty. The projected value has the same access control level as the original wrapped property.

1. @propertyWrapper
2. struct WrapperWithProjection {
3. var wrappedValue: Int
4. var projectedValue: SomeProjection {
5. return SomeProjection(wrapper: self)
6. }
7. }
8. struct SomeProjection {
9. var wrapper: WrapperWithProjection
10. }
11. struct SomeStruct {
12. @WrapperWithProjection var x = 123
13. }
14. let s = SomeStruct()
15. s.x // Int value
16. s.$x // SomeProjection value
17. s.$x.wrapper // WrapperWithProjection value

### **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(\_:\_:\_:\_:)](https://developer.apple.com/documentation/uikit/1622933-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. A structure or class marked usableFromInline can use only types that are public or usableFromInline for its properties. An enumeration marked usableFromInline can use only types that are public or usableFromInline for the raw values and associated values of its cases.

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(\_:\_:)](https://developer.apple.com/documentation/swift/1538339-min/) function and a [min()](https://developer.apple.com/documentation/swift/sequence/1641174-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](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) and [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449).

### **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](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546).

## 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](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602).

GRAMMAR OF AN ATTRIBUTE

# **Patterns**

A pattern represents the structure of a single value or a composite value. For example, the structure of a tuple (1, 2) is a comma-separated list of two elements. Because patterns represent the structure of a value rather than any one particular value, you can match them with a variety of values. For instance, the pattern (x, y) matches the tuple (1, 2) and any other two-element tuple. In addition to matching a pattern with a value, you can extract part or all of a composite value and bind each part to a constant or variable name.

In Swift, there are two basic kinds of patterns: those that successfully match any kind of value, and those that may fail to match a specified value at runtime.

The first kind of pattern is used for destructuring values in simple variable, constant, and optional bindings. These include wildcard patterns, identifier patterns, and any value binding or tuple patterns containing them. You can specify a type annotation for these patterns to constrain them to match only values of a certain type.

The second kind of pattern is used for full pattern matching, where the values you’re trying to match against may not be there at runtime. These include enumeration case patterns, optional patterns, expression patterns, and type-casting patterns. You use these patterns in a case label of a switch statement, a catch clause of a do statement, or in the case condition of an if, while, guard, or for-in statement.

GRAMMAR OF A PATTERN

*pattern* → [wildcard-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_wildcard-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [identifier-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_identifier-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [value-binding-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_value-binding-pattern)

*pattern* → [tuple-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [enum-case-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html" \l "grammar_enum-case-pattern)

*pattern* → [optional-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_optional-pattern)

*pattern* → [type-casting-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_type-casting-pattern)

*pattern* → [expression-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_expression-pattern)

## Wildcard Pattern

A wildcard pattern matches and ignores any value and consists of an underscore (\_). Use a wildcard pattern when you don’t care about the values being matched against. For example, the following code iterates through the closed range 1...3, ignoring the current value of the range on each iteration of the loop:

1. for \_ in 1...3 {
2. // Do something three times.
3. }

GRAMMAR OF A WILDCARD PATTERN

*wildcard-pattern* → **\_**

## Identifier Pattern

An identifier pattern matches any value and binds the matched value to a variable or constant name. For example, in the following constant declaration, someValue is an identifier pattern that matches the value 42 of type Int:

1. let someValue = 42

When the match succeeds, the value 42 is bound (assigned) to the constant name someValue.

When the pattern on the left-hand side of a variable or constant declaration is an identifier pattern, the identifier pattern is implicitly a subpattern of a value-binding pattern.

GRAMMAR OF AN IDENTIFIER PATTERN

*identifier-pattern* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

## Value-Binding Pattern

A value-binding pattern binds matched values to variable or constant names. Value-binding patterns that bind a matched value to the name of a constant begin with the let keyword; those that bind to the name of variable begin with the var keyword.

Identifiers patterns within a value-binding pattern bind new named variables or constants to their matching values. For example, you can decompose the elements of a tuple and bind the value of each element to a corresponding identifier pattern.

1. let point = (3, 2)
2. switch point {
3. // Bind x and y to the elements of point.
4. case let (x, y):
5. print("The point is at (\(x), \(y)).")
6. }
7. // Prints "The point is at (3, 2)."

In the example above, let distributes to each identifier pattern in the tuple pattern (x, y). Because of this behavior, the switch cases case let (x, y): and case (let x, let y): match the same values.

GRAMMAR OF A VALUE-BINDING PATTERN

*value-binding-pattern* → **var** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) | **let** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern)

## Tuple Pattern

A tuple pattern is a comma-separated list of zero or more patterns, enclosed in parentheses. Tuple patterns match values of corresponding tuple types.

You can constrain a tuple pattern to match certain kinds of tuple types by using type annotations. For example, the tuple pattern (x, y): (Int, Int) in the constant declaration let (x, y): (Int, Int) = (1, 2) matches only tuple types in which both elements are of type Int.

When a tuple pattern is used as the pattern in a for-in statement or in a variable or constant declaration, it can contain only wildcard patterns, identifier patterns, optional patterns, or other tuple patterns that contain those. For example, the following code isn’t valid because the element 0 in the tuple pattern (x, 0) is an expression pattern:

1. let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
2. // This code isn't valid.
3. for (x, 0) in points {
4. /\* ... \*/
5. }

The parentheses around a tuple pattern that contains a single element have no effect. The pattern matches values of that single element’s type. For example, the following are equivalent:

1. let a = 2 // a: Int = 2
2. let (a) = 2 // a: Int = 2
3. let (a): Int = 2 // a: Int = 2

GRAMMAR OF A TUPLE PATTERN

*tuple-pattern* → **(** [tuple-pattern-element-list](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element-list) *opt* **)**

*tuple-pattern-element-list* → [tuple-pattern-element](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element) | [tuple-pattern-element](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element) **,** [tuple-pattern-element-list](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element-list)

*tuple-pattern-element* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) | [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier) **:** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern)

## Enumeration Case Pattern

An enumeration case pattern matches a case of an existing enumeration type. Enumeration case patterns appear in switch statement case labels and in the case conditions of if, while, guard, and for-in statements.

If the enumeration case you’re trying to match has any associated values, the corresponding enumeration case pattern must specify a tuple pattern that contains one element for each associated value. For an example that uses a switch statement to match enumeration cases containing associated values, see [Associated Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID148).

An enumeration case pattern also matches values of that case wrapped in an optional. This simplified syntax lets you omit an optional pattern. Note that, because Optional is implemented as an enumeration, .none and .some can appear in the same switch as the cases of the enumeration type.

1. enum SomeEnum { case left, right }
2. let x: SomeEnum? = .left
3. switch x {
4. case .left:
5. print("Turn left")
6. case .right:
7. print("Turn right")
8. case nil:
9. print("Keep going straight")
10. }
11. // Prints "Turn left"

GRAMMAR OF AN ENUMERATION CASE PATTERN

*enum-case-pattern* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) *opt* **.** [enum-case-name](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_enum-case-name) [tuple-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern) *opt*

## Optional Pattern

An optional pattern matches values wrapped in a some(Wrapped) case of an Optional<Wrapped> enumeration. Optional patterns consist of an identifier pattern followed immediately by a question mark and appear in the same places as enumeration case patterns.

Because optional patterns are syntactic sugar for Optional enumeration case patterns, the following are equivalent:

1. let someOptional: Int? = 42
2. // Match using an enumeration case pattern.
3. if case .some(let x) = someOptional {
4. print(x)
5. }
6. // Match using an optional pattern.
7. if case let x? = someOptional {
8. print(x)
9. }

The optional pattern provides a convenient way to iterate over an array of optional values in a for-in statement, executing the body of the loop only for non-nil elements.

1. let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
2. // Match only non-nil values.
3. for case let number? in arrayOfOptionalInts {
4. print("Found a \(number)")
5. }
6. // Found a 2
7. // Found a 3
8. // Found a 5

GRAMMAR OF AN OPTIONAL PATTERN

*optional-pattern* → [identifier-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_identifier-pattern) **?**

## Type-Casting Patterns

There are two type-casting patterns, the is pattern and the as pattern. The is pattern appears only in switch statement case labels. The is and as patterns have the following form:

1. is type
2. pattern as type

The is pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the is pattern—or a subclass of that type. The is pattern behaves like the is operator in that they both perform a type cast but discard the returned type.

The as pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the as pattern—or a subclass of that type. If the match succeeds, the type of the matched value is cast to the pattern specified in the right-hand side of the as pattern.

For an example that uses a switch statement to match values with is and as patterns, see [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342).

GRAMMAR OF A TYPE CASTING PATTERN

*type-casting-pattern* → [is-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_is-pattern) | [as-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_as-pattern)

*is-pattern* → **is** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

*as-pattern* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) **as** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

## Expression Pattern

An expression pattern represents the value of an expression. Expression patterns appear only in switch statement case labels.

The expression represented by the expression pattern is compared with the value of an input expression using the Swift standard library ~= operator. The matches succeeds if the ~= operator returns true. By default, the ~= operator compares two values of the same type using the == operator. It can also match a value with a range of values, by checking whether the value is contained within the range, as the following example shows.

1. let point = (1, 2)
2. switch point {
3. case (0, 0):
4. print("(0, 0) is at the origin.")
5. case (-2...2, -2...2):
6. print("(\(point.0), \(point.1)) is near the origin.")
7. default:
8. print("The point is at (\(point.0), \(point.1)).")
9. }
10. // Prints "(1, 2) is near the origin."

You can overload the ~= operator to provide custom expression matching behavior. For example, you can rewrite the above example to compare the point expression with a string representations of points.

1. // Overload the ~= operator to match a string with an integer.
2. func ~= (pattern: String, value: Int) -> Bool {
3. return pattern == "\(value)"
4. }
5. switch point {
6. case ("0", "0"):
7. print("(0, 0) is at the origin.")
8. default:
9. print("The point is at (\(point.0), \(point.1)).")
10. }
11. // Prints "The point is at (1, 2)."

# **Generic Parameters and Arguments**

This chapter describes parameters and arguments for generic types, functions, and initializers. When you declare a generic type, function, subscript, or initializer, you specify the type parameters that the generic type, function, or initializer can work with. These type parameters act as placeholders that are replaced by actual concrete type arguments when an instance of a generic type is created or a generic function or initializer is called.

For an overview of generics in Swift, see [Generics](https://docs.swift.org/swift-book/LanguageGuide/Generics.html).

## Generic Parameter Clause

A generic parameter clause specifies the type parameters of a generic type or function, along with any associated constraints and requirements on those parameters. A generic parameter clause is enclosed in angle brackets (<>) and has the following form:

1. <generic parameter list>

The generic parameter list is a comma-separated list of generic parameters, each of which has the following form:

1. type parameter: constraint

A generic parameter consists of a type parameter followed by an optional constraint. A type parameter is simply the name of a placeholder type (for example, T, U, V, Key, Value, and so on). You have access to the type parameters (and any of their associated types) in the rest of the type, function, or initializer declaration, including in the signature of the function or initializer.

The constraint specifies that a type parameter inherits from a specific class or conforms to a protocol or protocol composition. For example, in the generic function below, the generic parameter T: Comparable indicates that any type argument substituted for the type parameter T must conform to the Comparable protocol.

1. func simpleMax<T: Comparable>(\_ x: T, \_ y: T) -> T {
2. if x < y {
3. return y
4. }
5. return x
6. }

Because Int and Double, for example, both conform to the Comparable protocol, this function accepts arguments of either type. In contrast with generic types, you don’t specify a generic argument clause when you use a generic function or initializer. The type arguments are instead inferred from the type of the arguments passed to the function or initializer.

1. simpleMax(17, 42) // T is inferred to be Int
2. simpleMax(3.14159, 2.71828) // T is inferred to be Double

### **Generic Where Clauses**

You can specify additional requirements on type parameters and their associated types by including a generic where clause right before the opening curly brace of a type or function’s body. A generic where clause consists of the where keyword, followed by a comma-separated list of one or more requirements.

1. where requirements

The requirements in a generic where clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the generic where clause provides syntactic sugar for expressing simple constraints on type parameters (for example, <T: Comparable> is equivalent to <T> where T: Comparable and so on), you can use it to provide more complex constraints on type parameters and their associated types. For example, you can constrain the associated types of type parameters to conform to protocols. For example, <S: Sequence> where S.Iterator.Element: Equatable specifies that S conforms to the Sequence protocol and that the associated type S.Iterator.Element conforms to the Equatable protocol. This constraint ensures that each element of the sequence is equatable.

You can also specify the requirement that two types be identical, using the == operator. For example, <S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element expresses the constraints that S1 and S2 conform to the Sequence protocol and that the elements of both sequences must be of the same type.

Any type argument substituted for a type parameter must meet all the constraints and requirements placed on the type parameter.

You can overload a generic function or initializer by providing different constraints, requirements, or both on the type parameters. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

For more information about generic where clauses and to see an example of one in a generic function declaration, see [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192).

GRAMMAR OF A GENERIC PARAMETER CLAUSE

*generic-parameter-clause* → **<** [generic-parameter-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter-list) **>**

*generic-parameter-list* → [generic-parameter](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter) | [generic-parameter](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter) **,** [generic-parameter-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter-list)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) **:** [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) **:** [protocol-composition-type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_protocol-composition-type)

*generic-where-clause* → **where** [requirement-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement-list)

*requirement-list* → [requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement) | [requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement) **,** [requirement-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement-list)

*requirement* → [conformance-requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_conformance-requirement) | [same-type-requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_same-type-requirement)

*conformance-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **:** [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier)

*conformance-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **:** [protocol-composition-type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_protocol-composition-type)

*same-type-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **==** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

## Generic Argument Clause

A generic argument clause specifies the type arguments of a generic type. A generic argument clause is enclosed in angle brackets (<>) and has the following form:

1. <generic argument list>

The generic argument list is a comma-separated list of type arguments. A type argument is the name of an actual concrete type that replaces a corresponding type parameter in the generic parameter clause of a generic type. The result is a specialized version of that generic type. The example below shows a simplified version of the Swift standard library’s generic dictionary type.

1. struct Dictionary<Key: Hashable, Value>: Collection, ExpressibleByDictionaryLiteral {
2. /\* ... \*/
3. }

The specialized version of the generic Dictionary type, Dictionary<String, Int> is formed by replacing the generic parameters Key: Hashable and Value with the concrete type arguments String and Int. Each type argument must satisfy all the constraints of the generic parameter it replaces, including any additional requirements specified in a generic where clause. In the example above, the Key type parameter is constrained to conform to the Hashable protocol and therefore String must also conform to the Hashable protocol.

You can also replace a type parameter with a type argument that is itself a specialized version of a generic type (provided it satisfies the appropriate constraints and requirements). For example, you can replace the type parameter Element in Array<Element> with a specialized version of an array, Array<Int>, to form an array whose elements are themselves arrays of integers.

1. let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

As mentioned in [Generic Parameter Clause](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#ID407), you don’t use a generic argument clause to specify the type arguments of a generic function or initializer.

# **Document Revision History**

**2019-09-10**

* Updated for Swift 5.1.
* Added information about functions that specify a protocol that their return value conforms to, instead of providing a specific named return type, to the [Opaque Types](https://docs.swift.org/swift-book/LanguageGuide/OpaqueTypes.html) chapter.
* Added information about property wrappers to the [Property Wrappers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617) section.
* Added information enumerations and structures that are frozen for library evolution to the [frozen](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID620) section.
* Added the [Functions With an Implicit Return](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID607) and [Shorthand Getter Declaration](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID608) sections with information about functions that omit return.
* Added information about using subscripts on types to the [Type Subscripts](https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html#ID609) section.
* Updated the [Enumeration Case Pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#ID424) section, now that an enumeration case pattern can match an optional value.
* Updated the [Memberwise Initializers for Structure Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID214) section, now that memberwise initializers support omitting parameters for properties that have a default value.
* Added information about dynamic members that are looked up by key path at run time to the [dynamicMemberLookup](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID585) section.
* Added macCatalyst to the list of target environments in [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539).
* Updated the [Self Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID610) section, now that Self can be used to refer to the type introduced by the current class, structure, or enumeration declaration.

**2019-03-25**

* Updated for Swift 5.0.
* Added the [Extended String Delimiters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID606) section and updated the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section with information about extended string delimiters.
* Added the [dynamicCallable](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID603) section with information about dynamically calling instances as functions using the dynamicCallable attribute.
* Added the [unknown](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID605) and [Switching Over Future Enumeration Cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602) sections with information about handling future enumeration cases in switch statements using the unknown switch case attribute.
* Added information about the identity key path (\.self) to the [Key-Path Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID563) section.
* Added information about using the less than (<) operator in platform conditions to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.

**2018-09-17**

* Updated for Swift 4.2.
* Added information about accessing all of an enumeration’s cases to the [Iterating over Enumeration Cases](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID581) section.
* Added information about #error and #warning to the [Compile-Time Diagnostic Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID582) section.
* Added information about inlining to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section under the inlinable and usableFromInline attributes.
* Added information about members that are looked up by name at runtime to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section under the dynamicMemberLookup attribute.
* Added information about the requires\_stored\_property\_inits and warn\_unqualified\_access attributes to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* Added information about how to conditionally compile code depending on the Swift compiler version being used to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.
* Added information about #dsohandle to the [Literal Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID390) section.

**2018-03-29**

* Updated for Swift 4.1.
* Added information about synthesized implementations of equivalence operators to the [Equivalence Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID45) section.
* Added information about conditional protocol conformance to the [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter, and to the [Conditionally Conforming to a Protocol](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID574) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter.
* Added information about recursive protocol constraints to the [Using a Protocol in Its Associated Type’s Constraints](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID575) section.
* Added information about the canImport() and targetEnvironment() platform conditions to [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539).

**2017-12-04**

* Updated for Swift 4.0.3.
* Updated the [Key-Path Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID563) section, now that key paths support subscript components.

**2017-09-19**

* Updated for Swift 4.0.
* Added information about exclusive access to memory to the [Memory Safety](https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html) chapter.
* Added the [Associated Types with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID557) section, now that you can use generic where clauses to constrain associated types.
* Added information about multiline string literals to the [String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID286) section of the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter, and to the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section of the [Lexical Structure](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html) chapter.
* Updated the discussion of the objc attribute in [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348), now that this attribute is inferred in fewer places.
* Added the [Generic Subscripts](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID558) section, now that subscripts can be generic.
* Updated the discussion in the [Protocol Composition](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID282) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter, and in the [Protocol Composition Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID454) section of the [Types](https://docs.swift.org/swift-book/ReferenceManual/Types.html) chapter, now that protocol composition types can contain a superclass requirement.
* Updated the discussion of protocol extensions in [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378) now that final isn’t allowed in them.
* Added information about preconditions and fatal errors to the [Assertions and Preconditions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335) section.

**2017-03-27**

* Updated for Swift 3.1.
* Added the [Extensions with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID553) section with information about extensions that include requirements.
* Added examples of iterating over a range to the [For-In Loops](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID121) section.
* Added an example of failable numeric conversions to the [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224) section.
* Added information to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section about using the available attribute with a Swift language version.
* Updated the discussion in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section to note that argument labels are not allowed when writing a function type.
* Updated the discussion of Swift language version numbers in the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section, now that an optional patch number is allowed.
* Updated the discussion in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section, now that Swift distinguishes between functions that take multiple parameters and functions that take a single parameter of a tuple type.
* Removed the Dynamic Type Expression section from the [Expressions](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html) chapter, now that type(of:) is a Swift standard library function.

**2016-10-27**

* Updated for Swift 3.0.1.
* Updated the discussion of weak and unowned references in the [Automatic Reference Counting](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html) chapter.
* Added information about the unowned, unowned(safe), and unowned(unsafe) declaration modifiers in the [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381) section.
* Added a note to the [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342) section about using an optional value when a value of type Any is expected.
* Updated the [Expressions](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html) chapter to separate the discussion of parenthesized expressions and tuple expressions.

**2016-09-13**

* Updated for Swift 3.0.
* Updated the discussion of functions in the [Functions](https://docs.swift.org/swift-book/LanguageGuide/Functions.html) chapter and the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section to note that all parameters get an argument label by default.
* Updated the discussion of operators in the [Advanced Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html) chapter, now that you implement them as type methods instead of as global functions.
* Added information about the open and fileprivate access-level modifiers to the [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html) chapter.
* Updated the discussion of inout in the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section to note that it appears in front of a parameter’s type instead of in front of a parameter’s name.
* Updated the discussion of the @noescape and @autoclosure attributes in the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) and [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) sections and the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter now that they are type attributes, rather than declaration attributes.
* Added information about operator precedence groups to the [Precedence for Custom Infix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID47) section of the [Advanced Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html) chapter, and to the [Precedence Group Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID550) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Updated discussion throughout to use macOS instead of OS X, Error instead of ErrorProtocol, and protocol names such as ExpressibleByStringLiteral instead of StringLiteralConvertible.
* Updated the discussion in the [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192) section of the [Generics](https://docs.swift.org/swift-book/LanguageGuide/Generics.html) chapter and in the [Generic Parameters and Arguments](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html) chapter, now that generic where clauses are written at the end of a declaration.
* Updated the discussion in the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) section, now that closures are nonescaping by default.
* Updated the discussion in the [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333) section of the [The Basics](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html) chapter and the [While Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID432) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter, now that if, while, and guard statements use a comma-separated list of conditions without where clauses.
* Added information about switch cases that have multiple patterns to the [Switch](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID129) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Switch Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID436) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Updated the discussion of function types in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section now that function argument labels are no longer part of a function’s type.
* Updated the discussion of protocol composition types in the [Protocol Composition](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID282) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter and in the [Protocol Composition Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID454) section of the [Types](https://docs.swift.org/swift-book/ReferenceManual/Types.html) chapter to use the new Protocol1 & Protocol2 syntax.
* Updated the discussion in the Dynamic Type Expression section to use the new type(of:) syntax for dynamic type expressions.
* Updated the discussion of line control statements to use the #sourceLocation(file:line:) syntax in the [Line Control Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID540) section.
* Updated the discussion in [Functions that Never Return](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID551) to use the new Never type.
* Added information about playground literals to the [Literal Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID390) section.
* Updated the discussion in the [In-Out Parameters](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545) section to note that only nonescaping closures can capture in-out parameters.
* Updated the discussion about default parameters in the [Default Parameter Values](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID169) section, now that they can’t be reordered in function calls.
* Updated attribute arguments to use a colon in the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter.
* Added information about throwing an error inside the catch block of a rethrowing function to the [Rethrowing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID531) section.
* Added information about accessing the selector of an Objective-C property’s getter or setter to the [Selector Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID547) section.
* Added information to the [Type Alias Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID361) section about generic type aliases and using type aliases inside of protocols.
* Updated the discussion of function types in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section to note that parentheses around the parameter types are required.
* Updated the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter to note that the @IBAction, @IBOutlet, and @NSManaged attributes imply the @objc attribute.
* Added information about the @GKInspectable attribute to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* Updated the discussion of optional protocol requirements in the [Optional Protocol Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID284) section to clarify that they are used only in code that interoperates with Objective-C.
* Removed the discussion of explicitly using let in function parameters from the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section.
* Removed the discussion of the Boolean protocol from the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter, now that the protocol has been removed from the Swift standard library.
* Corrected the discussion of the @NSApplicationMain attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.

**2016-03-21**

* Updated for Swift 2.2.
* Added information about how to conditionally compile code depending on the version of Swift being used to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.
* Added information about how to distinguish between methods or initializers whose names differ only by the names of their arguments to the [Explicit Member Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID400) section.
* Added information about the #selector syntax for Objective-C selectors to the [Selector Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID547) section.
* Updated the discussion of associated types to use the associatedtype keyword in the [Associated Types](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID189) and [Protocol Associated Type Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID374) sections.
* Updated information about initializers that return nil before the instance is fully initialized in the [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224) section.
* Added information about comparing tuples to the [Comparison Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID70) section.
* Added information about using keywords as external parameter names to the [Keywords and Punctuation](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID413) section.
* Updated the discussion of the @objc attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section to note that enumerations and enumeration cases can use this attribute.
* Updated the [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) section with discussion of custom operators that contain a dot.
* Added a note to the [Rethrowing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID531) section that rethrowing functions can’t directly throw errors.
* Added a note to the [Property Observers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262) section about property observers being called when you pass a property as an in-out parameter.
* Added a section about error handling to the [A Swift Tour](https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html) chapter.
* Updated figures in the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section to show the deallocation process more clearly.
* Removed discussion of C-style for loops, the ++ prefix and postfix operators, and the -- prefix and postfix operators.
* Removed discussion of variable function arguments and the special syntax for curried functions.

**2015-10-20**

* Updated for Swift 2.1.
* Updated the [String Interpolation](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID292) and [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) sections now that string interpolations can contain string literals.
* Added the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) section with information about the @noescape attribute.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) and [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) sections with information about tvOS.
* Added information about the behavior of in-out parameters to the [In-Out Parameters](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545) section.
* Added information to the [Capture Lists](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID544) section about how values specified in closure capture lists are captured.
* Updated the [Accessing Properties Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID248) section to clarify how assignment through optional chaining behaves.
* Improved the discussion of autoclosures in the [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) section.
* Added an example that uses the ?? operator to the [A Swift Tour](https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html) chapter.

**2015-09-16**

* Updated for Swift 2.0.
* Added information about error handling to the [Error Handling](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html) chapter, the [Do Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID533) section, the [Throw Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID518) section, the [Defer Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID532) section, and the [Try Operator](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID516) section.
* Updated the [Representing and Throwing Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID509) section, now that all types can conform to the ErrorType protocol.
* Added information about the new try? keyword to the [Converting Errors to Optional Values](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID542) section.
* Added information about recursive enumerations to the [Recursive Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID536) section of the [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html) chapter and the [Enumerations with Cases of Any Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID365) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Added information about API availability checking to the [Checking API Availability](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID523) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Availability Condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID522) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Added information about the new guard statement to the [Early Exit](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID525) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Guard Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID524) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Added information about protocol extensions to the [Protocol Extensions](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID521) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter.
* Added information about access control for unit testing to the [Access Levels for Unit Test Targets](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID519) section of the [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html) chapter.
* Added information about the new optional pattern to the [Optional Pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#ID520) section of the [Patterns](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html) chapter.
* Updated the [Repeat-While](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID126) section with information about the repeat-while loop.
* Updated the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter, now that String no longer conforms to the CollectionType protocol from the Swift standard library.
* Added information about the new Swift standard library print(\_:separator:terminator) function to the [Printing Constants and Variables](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID314) section.
* Added information about the behavior of enumeration cases with String raw values to the [Implicitly Assigned Raw Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID535) section of the [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html) chapter and the [Enumerations with Cases of a Raw-Value Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID366) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Added information about the @autoclosure attribute—including its @autoclosure(escaping) form—to the [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) section.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section with information about the @available and @warn\_unused\_result attributes.
* Updated the [Type Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID350) section with information about the @convention attribute.
* Added an example of using multiple optional bindings with a where clause to the [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333) section.
* Added information to the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section about how concatenating string literals using the + operator happens at compile time.
* Added information to the [Metatype Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html" \l "ID455) section about comparing metatype values and using them to construct instances with initializer expressions.
* Added a note to the [Debugging with Assertions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID336) section about when user-defined assertions are disabled.
* Updated the discussion of the @NSManaged attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section, now that the attribute can be applied to certain instance methods.
* Updated the [Variadic Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID171) section, now that variadic parameters can be declared in any position in a function’s parameter list.
* Added information to the [Overriding a Failable Initializer](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID229) section about how a nonfailable initializer can delegate up to a failable initializer by force-unwrapping the result of the superclass’s initializer.
* Added information about using enumeration cases as functions to the [Enumerations with Cases of Any Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID365) section.
* Added information about explicitly referencing an initializer to the [Initializer Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID399) section.
* Added information about build configuration and line control statements to the [Compiler Control Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID538) section.
* Added a note to the [Metatype Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html" \l "ID455) section about constructing class instances from metatype values.
* Added a note to the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section about weak references being unsuitable for caching.
* Updated a note in the [Type Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264) section to mention that stored type properties are lazily initialized.
* Updated the [Capturing Values](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID103) section to clarify how variables and constants are captured in closures.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section to describe when you can apply the @objc attribute to classes.
* Added a note to the [Handling Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID512) section about the performance of executing a throw statement. Added similar information about the do statement in the [Do Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID533) section.
* Updated the [Type Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264) section with information about stored and computed type properties for classes, structures, and enumerations.
* Updated the [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) section with information about labeled break statements.
* Updated a note in the [Property Observers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262) section to clarify the behavior of willSet and didSet observers.
* Added a note to the [Access Levels](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID5) section with information about the scope of private access.
* Added a note to the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section about the differences in weak references between garbage collected systems and ARC.
* Updated the [Special Characters in String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID295) section with a more precise definition of Unicode scalars.

**2015-04-08**

* Updated for Swift 1.2.
* Swift now has a native Set collection type. For more information, see [Sets](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID484).
* @autoclosure is now an attribute of the parameter declaration, not its type. There is also a new @noescape parameter declaration attribute. For more information, see [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348).
* Type methods and properties now use the static keyword as a declaration modifier. For more information see [Type Variable Properties](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID483).
* Swift now includes the as? and as! failable downcast operators. For more information, see [Checking for Protocol Conformance](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID283).
* Added a new guide section about [String Indices](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID534).
* Removed the overflow division (&/) and overflow remainder (&%) operators from [Overflow Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID37).
* Updated the rules for constant and constant property declaration and initialization. For more information, see [Constant Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID355).
* Updated the definition of Unicode scalars in string literals. See [Special Characters in String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID295).
* Updated [Range Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID73) to note that a half-open range with the same start and end index will be empty.
* Updated [Closures Are Reference Types](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID104) to clarify the capturing rules for variables.
* Updated [Value Overflow](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID38) to clarify the overflow behavior of signed and unsigned integers
* Updated [Protocol Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID369) to clarify protocol declaration scope and members.
* Updated [Defining a Capture List](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID58) to clarify the syntax for weak and unowned references in closure capture lists.
* Updated [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) to explicitly mention examples of supported characters for custom operators, such as those in the Mathematical Operators, Miscellaneous Symbols, and Dingbats Unicode blocks.
* Constants can now be declared without being initialized in local function scope. They must have a set value before first use. For more information, see [Constant Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID355).
* In an initializer, constant properties can now only assign a value once. For more information, see [Assigning Constant Properties During Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID212).
* Multiple optional bindings can now appear in a single if statement as a comma-separated list of assignment expressions. For more information, see [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).
* An [Optional-Chaining Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID405) must appear within a postfix expression.
* Protocol casts are no longer limited to @objc protocols.
* Type casts that can fail at runtime now use the as? or as! operator, and type casts that are guaranteed not to fail use the as operator. For more information, see [Type-Casting Operators](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID388).

**2014-10-16**

* Updated for Swift 1.1.
* Added a full guide to [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224).
* Added a description of [Failable Initializer Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html" \l "ID274) for protocols.
* Constants and variables of type Any can now contain function instances. Updated the example in [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342) to show how to check for and cast to a function type within a switch statement.
* Enumerations with raw values now have a rawValue property rather than a toRaw() method and a failable initializer with a rawValue parameter rather than a fromRaw() method. For more information, see [Raw Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID149) and [Enumerations with Cases of a Raw-Value Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID366).
* Added a new reference section about [Failable Initializers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html" \l "ID376), which can trigger initialization failure.
* Custom operators can now contain the ? character. Updated the [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) reference to describe the revised rules. Removed a duplicate description of the valid set of operator characters from [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46).

**2014-08-18**

* New document that describes Swift 1.0, Apple’s new programming language for building iOS and OS X apps.
* Added a new section about [Initializer Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID272) in protocols.
* Added a new section about [Class-Only Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID281).
* [Assertions and Preconditions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335) can now use string interpolation. Removed a note to the contrary.
* Updated the [Concatenating Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID291) section to reflect the fact that String and Character values can no longer be combined with the addition operator (+) or addition assignment operator (+=). These operators are now used only with String values. Use the String type’s append(\_:) method to append a single Character value onto the end of a string.
* Added information about the availability attribute to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* [Optionals](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID330) no longer implicitly evaluate to true when they have a value and false when they do not, to avoid confusion when working with optional Bool values. Instead, make an explicit check against nil with the == or != operators to find out if an optional contains a value.
* Swift now has a [Nil-Coalescing Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID72) (a ?? b), which unwraps an optional’s value if it exists, or returns a default value if the optional is nil.
* Updated and expanded the [Comparing Strings](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID298) section to reflect and demonstrate that string and character comparison and prefix / suffix comparison are now based on Unicode canonical equivalence of extended grapheme clusters.
* You can now try to set a property’s value, assign to a subscript, or call a mutating method or operator through [Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html). The information about [Accessing Properties Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID248) has been updated accordingly, and the examples of checking for method call success in [Calling Methods Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID249) have been expanded to show how to check for property setting success.
* Added a new section about [Accessing Subscripts of Optional Type](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID251) through optional chaining.
* Updated the [Accessing and Modifying an Array](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID110) section to note that you can no longer append a single item to an array with the += operator. Instead, use the append(\_:) method, or append a single-item array with the += operator.
* Added a note that the start value a for the [Range Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID73) a...b and a..<b must not be greater than the end value b.
* Rewrote the [Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html) chapter to remove its introductory coverage of initializer overrides. This chapter now focuses more on the addition of new functionality in a subclass, and the modification of existing functionality with overrides. The chapter’s example of [Overriding Property Getters and Setters](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html#ID200) has been rewritten to show how to override a description property. (The examples of modifying an inherited property’s default value in a subclass initializer have been moved to the [Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html) chapter.)
* Updated the [Initializer Inheritance and Overriding](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID221) section to note that overrides of a designated initializer must now be marked with the override modifier.
* Updated the [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231) section to note that the required modifier is now written before every subclass implementation of a required initializer, and that the requirements for required initializers can now be satisfied by automatically inherited initializers.
* Infix [Operator Methods](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID42) no longer require the @infix attribute.
* The @prefix and @postfix attributes for [Prefix and Postfix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID43) have been replaced by prefix and postfix declaration modifiers.
* Added a note about the order in which [Prefix and Postfix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID43) are applied when both a prefix and a postfix operator are applied to the same operand.
* Operator functions for [Compound Assignment Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID44) no longer use the @assignment attribute when defining the function.
* The order in which modifiers are specified when defining [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46) has changed. You now write prefix operator rather than operator prefix, for example.
* Added information about the dynamic declaration modifier in [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381).
* Added information about how type inference works with [Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID414).
* Added more information about curried functions.
* Added a new chapter about [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html).
* Updated the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter to reflect the fact that Swift’s Character type now represents a single Unicode extended grapheme cluster. Includes a new section on [Extended Grapheme Clusters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID296) and more information about [Unicode Scalar Values](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID294) and [Comparing Strings](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID298).
* Updated the [String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID286) section to note that Unicode scalars inside string literals are now written as \u{n}, where n is a hexadecimal number between 0 and 10FFFF, the range of Unicode’s codespace.
* The NSString length property is now mapped onto Swift’s native String type as utf16Count, not utf16count.
* Swift’s native String type no longer has an uppercaseString or lowercaseString property. The corresponding section in [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) has been removed, and various code examples have been updated.
* Added a new section about [Initializer Parameters Without Argument Labels](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID210).
* Added a new section about [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231).
* Added a new section about [Optional Tuple Return Types](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID165).
* Updated the [Type Annotations](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID312) section to note that multiple related variables can be defined on a single line with one type annotation.
* The @optional, @lazy, @final, and @required attributes are now the optional, lazy, final, and required [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381).
* Updated the entire book to refer to ..< as the [Half-Open Range Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID75) (rather than the “half-closed range operator”).
* Updated the [Accessing and Modifying a Dictionary](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID116) section to note that Dictionary now has a Boolean isEmpty property.
* Clarified the full list of characters that can be used when defining [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46).
* nil and the Booleans true and false are now [Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID414).
* Swift’s Array type now has full value semantics. Updated the information about [Mutability of Collections](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID106) and [Arrays](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID107) to reflect the new approach. Also clarified the assignment and copy behavior for strings arrays and dictionaries.
* [Array Type Shorthand Syntax](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID108) is now written as [SomeType] rather than SomeType[].
* Added a new section about [Dictionary Type Shorthand Syntax](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID114), which is written as [KeyType: ValueType].
* Added a new section about [Hash Values for Set Types](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID493).
* Examples of [Closure Expressions](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID95) now use the global sorted(\_:\_:) function rather than the global sort(\_:\_:) function, to reflect the new array value semantics.
* Updated the information about [Memberwise Initializers for Structure Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID214) to clarify that the memberwise structure initializer is made available even if a structure’s stored properties do not have default values.
* Updated to ..< rather than .. for the [Half-Open Range Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID75).
* Added an example of [Extending a Generic Type](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID185).

Keywords and Punctuation

The following keywords are reserved and can’t be used as identifiers, unless they’re escaped with backticks, as described above in [Identifiers](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID412). Keywords other than inout, var, and let can be used as parameter names in a function declaration or function call without being escaped with backticks. When a member has the same name as a keyword, references to that member don’t need to be escaped with backticks, except when there’s ambiguity between referring to the member and using the keyword—for example, self, Type, and Protocol have special meaning in an explicit member expression, so they must be escaped with backticks in that context.

* Keywords used in declarations: associatedtype, class, deinit, enum, extension, fileprivate, func, import, init, inout, internal, let, open, operator, private, protocol, public, rethrows, static, struct, subscript, typealias, and var.
* Keywords used in statements: break, case, continue, default, defer, do, else, fallthrough, for, guard, if, in, repeat, return, switch, where, and while.
* Keywords used in expressions and types: as, Any, catch, false, is, nil, super, self, Self, throw, throws, true, and try.
* Keywords used in patterns: \_.
* Keywords that begin with a number sign (#): #available, #colorLiteral, #column, #else, #elseif, #endif, #error, #file, #fileLiteral, #function, #if, #imageLiteral, #line, #selector, #sourceLocation, and #warning.
* Keywords reserved in particular contexts: associativity, convenience, dynamic, didSet, final, get, infix, indirect, lazy, left, mutating, none, nonmutating, optional, override, postfix, precedence, prefix, Protocol, required, right, set, Type, unowned, weak, and willSet. Outside the context in which they appear in the grammar, they can be used as identifiers.

The following tokens are reserved as punctuation and can’t be used as custom operators: (, ), {, }, [, ], ., ,, :, ;, =, @, #, & (as a prefix operator), ->, `, ?, and ! (as a postfix operator).

Literals

A *literal* is the source code representation of a value of a type, such as a number or string.

The following are examples of literals:

1. 42 // Integer literal
2. 3.14159 // Floating-point literal
3. "Hello, world!" // String literal
4. true // Boolean literal

A literal doesn’t have a type on its own. Instead, a literal is parsed as having infinite precision and Swift’s type inference attempts to infer a type for the literal. For example, in the declaration let x: Int8 = 42, Swift uses the explicit type annotation (: Int8) to infer that the type of the integer literal 42 is Int8. If there isn’t suitable type information available, Swift infers that the literal’s type is one of the default literal types defined in the Swift standard library. The default types are Int for integer literals, Double for floating-point literals, String for string literals, and Bool for Boolean literals. For example, in the declaration let str = "Hello, world", the default inferred type of the string literal "Hello, world" is String.

When specifying the type annotation for a literal value, the annotation’s type must be a type that can be instantiated from that literal value. That is, the type must conform to one of the following Swift standard library protocols: ExpressibleByIntegerLiteral for integer literals, ExpressibleByFloatLiteral for floating-point literals, ExpressibleByStringLiteral for string literals, ExpressibleByBooleanLiteral for Boolean literals, ExpressibleByUnicodeScalarLiteral for string literals that contain only a single Unicode scalar, and ExpressibleByExtendedGraphemeClusterLiteral for string literals that contain only a single extended grapheme cluster. For example, Int8 conforms to the ExpressibleByIntegerLiteral protocol, and therefore it can be used in the type annotation for the integer literal 42 in the declaration let x: Int8 = 42.

### **Integer Literals**

Integer literals represent integer values of unspecified precision. By default, integer literals are expressed in decimal; you can specify an alternate base using a prefix. Binary literals begin with 0b, octal literals begin with 0o, and hexadecimal literals begin with 0x.

Decimal literals contain the digits 0 through 9. Binary literals contain 0 and 1, octal literals contain 0 through 7, and hexadecimal literals contain 0 through 9 as well as A through F in upper- or lowercase.

Negative integers literals are expressed by prepending a minus sign (-) to an integer literal, as in -42.

Underscores (\_) are allowed between digits for readability, but they’re ignored and therefore don’t affect the value of the literal. Integer literals can begin with leading zeros (0), but they’re likewise ignored and don’t affect the base or value of the literal.

Unless otherwise specified, the default inferred type of an integer literal is the Swift standard library type Int. The Swift standard library also defines types for various sizes of signed and unsigned integers, as described in [Integers](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID317).

### **Floating-Point Literals**

Floating-point literals represent floating-point values of unspecified precision.

By default, floating-point literals are expressed in decimal (with no prefix), but they can also be expressed in hexadecimal (with a 0x prefix).

Decimal floating-point literals consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point (.) followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase e prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the e is multiplied by. For example, 1.25e2 represents 1.25 x 102, which evaluates to 125.0. Similarly, 1.25e-2 represents 1.25 x 10-2, which evaluates to 0.0125.

Hexadecimal floating-point literals consist of a 0x prefix, followed by an optional hexadecimal fraction, followed by a hexadecimal exponent. The hexadecimal fraction consists of a decimal point followed by a sequence of hexadecimal digits. The exponent consists of an upper- or lowercase p prefix followed by a sequence of decimal digits that indicates what power of 2 the value preceding the p is multiplied by. For example, 0xFp2 represents 15 x 22, which evaluates to 60. Similarly, 0xFp-2 represents 15 x 2-2, which evaluates to 3.75.

Negative floating-point literals are expressed by prepending a minus sign (-) to a floating-point literal, as in -42.5.

Underscores (\_) are allowed between digits for readability, but they’re ignored and therefore don’t affect the value of the literal. Floating-point literals can begin with leading zeros (0), but they’re likewise ignored and don’t affect the base or value of the literal.

Unless otherwise specified, the default inferred type of a floating-point literal is the Swift standard library type Double, which represents a 64-bit floating-point number. The Swift standard library also defines a Float type, which represents a 32-bit floating-point number.

### **String Literals**

A string literal is a sequence of characters surrounded by quotation marks. A single-line string literal is surrounded by double quotation marks and has the following form:

1. "characters"

String literals can’t contain an unescaped double quotation mark ("), an unescaped backslash (\), a carriage return, or a line feed.

A multiline string literal is surrounded by three double quotation marks and has the following form:

1. """
2. characters
3. """

Unlike a single-line string literal, a multiline string literal can contain unescaped double quotation marks ("), carriage returns, and line feeds. It can’t contain three unescaped double quotation marks next to each other.

The line break after the """ that begins the multiline string literal is not part of the string. The line break before the """ that ends the literal is also not part of the string. To make a multiline string literal that begins or ends with a line feed, write a blank line as its first or last line.

A multiline string literal can be indented using any combination of spaces and tabs; this indentation is not included in the string. The """ that ends the literal determines the indentation: Every nonblank line in the literal must begin with exactly the same indentation that appears before the closing """; there’s no conversion between tabs and spaces. You can include additional spaces and tabs after that indentation; those spaces and tabs appear in the string.

Line breaks in a multiline string literal are normalized to use the line feed character. Even if your source file has a mix of carriage returns and line feeds, all of the line breaks in the string will be the same.

In a multiline string literal, writing a backslash (\) at the end of a line omits that line break from the string. Any whitespace between the backslash and the line break is also omitted. You can use this syntax to hard wrap a multiline string literal in your source code, without changing the value of the resulting string.

Special characters can be included in string literals of both the single-line and multiline forms using the following escape sequences:

* Null character (\0)
* Backslash (\\)
* Horizontal tab (\t)
* Line feed (\n)
* Carriage return (\r)
* Double quotation mark (\")
* Single quotation mark (\')
* Unicode scalar (\u{n}), where n is a hexadecimal number that has one to eight digits

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (\). The interpolated expression can contain a string literal, but can’t contain an unescaped backslash, a carriage return, or a line feed.

For example, all of the following string literals have the same value:

1. "1 2 3"
2. "1 2 \("3")"
3. "1 2 \(3)"
4. "1 2 \(1 + 2)"
5. let x = 3; "1 2 \(x)"

A string delimited by extended delimiters is a sequence of characters surrounded by quotation marks and a balanced set of one or more number signs (#). A string delimited by extended delimiters has the following forms:

1. #"characters"#
2. #"""
3. characters
4. """#

Special characters in a string delimited by extended delimiters appear in the resulting string as normal characters rather than as special characters. You can use extended delimiters to create strings with characters that would ordinarily have a special effect such as generating a string interpolation, starting an escape sequence, or terminating the string.

The following example shows a string literal and a string delimited by extended delimiters that create equivalent string values:

1. let string = #"\(x) \ " \u{2603}"#
2. let escaped = "\\(x) \\ \" \\u{2603}"
3. print(string)
4. // Prints "\(x) \ " \u{2603}"
5. print(string == escaped)
6. // Prints "true"

If you use more than one number sign to form a string delimited by extended delimiters, don’t place whitespace in between the number signs:

1. print(###"Line 1\###nLine 2"###) // OK
2. print(# # #"Line 1\# # #nLine 2"# # #) // Error

Multiline string literals that you create using extended delimiters have the same indentation requirements as regular multiline string literals.

The default inferred type of a string literal is String. For more information about the String type, see [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) and [String](https://developer.apple.com/documentation/swift/string).

String literals that are concatenated by the + operator are concatenated at compile time. For example, the values of textA and textB in the example below are identical—no runtime concatenation is performed.

1. let textA = "Hello " + "world"
2. let textB = "Hello world"

# **Types**

In Swift, there are two kinds of types: named types and compound types. A named type is a type that can be given a particular name when it’s defined. Named types include classes, structures, enumerations, and protocols. For example, instances of a user-defined class named MyClass have the type MyClass. In addition to user-defined named types, the Swift standard library defines many commonly used named types, including those that represent arrays, dictionaries, and optional values.

Data types that are normally considered basic or primitive in other languages—such as types that represent numbers, characters, and strings—are actually named types, defined and implemented in the Swift standard library using structures. Because they’re named types, you can extend their behavior to suit the needs of your program, using an extension declaration, discussed in [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html) and [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378).

A compound type is a type without a name, defined in the Swift language itself. There are two compound types: function types and tuple types. A compound type may contain named types and other compound types. For example, the tuple type (Int, (Int, Int)) contains two elements: The first is the named type Int, and the second is another compound type (Int, Int).

You can put parentheses around a named type or a compound type. However, adding parentheses around a type doesn’t have any effect. For example, (Int) is equivalent to Int.

## Type Annotation

A type annotation explicitly specifies the type of a variable or expression. Type annotations begin with a colon (:) and end with a type, as the following examples show:

1. let someTuple: (Double, Double) = (3.14159, 2.71828)
2. func someFunction(a: Int) { /\* ... \*/ }

In the first example, the expression someTuple is specified to have the tuple type (Double, Double). In the second example, the parameter a to the function someFunction is specified to have the type Int.

Type annotations can contain an optional list of type attributes before the type

## Metatype Type

A metatype type refers to the type of any type, including class types, structure types, enumeration types, and protocol types.

The metatype of a class, structure, or enumeration type is the name of that type followed by .Type. The metatype of a protocol type—not the concrete type that conforms to the protocol at runtime—is the name of that protocol followed by .Protocol. For example, the metatype of the class type SomeClass is SomeClass.Type and the metatype of the protocol SomeProtocol is SomeProtocol.Protocol.

You can use the postfix self expression to access a type as a value. For example, SomeClass.self returns SomeClass itself, not an instance of SomeClass. And SomeProtocol.self returns SomeProtocol itself, not an instance of a type that conforms to SomeProtocol at runtime. You can call the type(of:) function with an instance of a type to access that instance’s dynamic, runtime type as a value, as the following example shows:

1. class SomeBaseClass {
2. class func printClassName() {
3. print("SomeBaseClass")
4. }
5. }
6. class SomeSubClass: SomeBaseClass {
7. override class func printClassName() {
8. print("SomeSubClass")
9. }
10. }
11. let someInstance: SomeBaseClass = SomeSubClass()
12. // The compile-time type of someInstance is SomeBaseClass,
13. // and the runtime type of someInstance is SomeSubClass
14. type(of: someInstance).printClassName()
15. // Prints "SomeSubClass"

For more information, see [type(of:)](https://developer.apple.com/documentation/swift/2885064-type) in the Swift standard library.

Use an initializer expression to construct an instance of a type from that type’s metatype value. For class instances, the initializer that’s called must be marked with the required keyword or the entire class marked with the final keyword.

1. class AnotherSubClass: SomeBaseClass {
2. let string: String
3. required init(string: String) {
4. self.string = string
5. }
6. override class func printClassName() {
7. print("AnotherSubClass")
8. }
9. }
10. let metatype: AnotherSubClass.Type = AnotherSubClass.self
11. let anotherInstance = metatype.init(string: "some string")

GRAMMAR OF A METATYPE TYPE

*metatype-type* → [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) **.** **Type** | [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) **.** **Protocol**

## Self Type

The Self type isn’t a specific type, but rather lets you conveniently refer to the current type without repeating or knowing that type’s name.

In a protocol declaration or a protocol member declaration, the Self type refers to the eventual type that conforms to the protocol.

In a structure, class, or enumeration declaration, the Self type refers to the type introduced by the declaration. Inside the declaration for a member of a type, the Self type refers to that type. In the members of a class declaration, Self can appear as the return type of a method and in the body of a method, but not in any other context. For example, the code below shows an instance method f whose return type is Self.

1. class Superclass {
2. func f() -> Self { return self }
3. }
4. let x = Superclass()
5. print(type(of: x.f()))
6. // Prints "Superclass"
7. class Subclass: Superclass { }
8. let y = Subclass()
9. print(type(of: y.f()))
10. // Prints "Subclass"
11. let z: Superclass = Subclass()
12. print(type(of: z.f()))
13. // Prints "Subclass"

The last part of the example above shows that Self refers to the runtime type Subclass of the value of z, not the compile-time type Superclass of the variable itself.

Inside a nested type declaration, the Self type refers to the type introduced by the innermost type declaration.

The Self type refers to the same type as the [type(of:)](https://developer.apple.com/documentation/swift/2885064-type) function in the Swift standard library. Writing Self.someStaticMember to access a member of the current type is the same as writing type(of: self).someStaticMember.

### **Try Operator**

A try expression consists of the try operator followed by an expression that can throw an error. It has the following form:

1. try expression

An optional-try expression consists of the try? operator followed by an expression that can throw an error. It has the following form:

1. try? expression

If the expression does not throw an error, the value of the optional-try expression is an optional containing the value of the expression. Otherwise, the value of the optional-try expression is nil.

A forced-try expression consists of the try! operator followed by an expression that can throw an error. It has the following form:

1. try! expression

If the expression throws an error, a runtime error is produced.

When the expression on the left-hand side of a binary operator is marked with try, try?, or try!, that operator applies to the whole binary expression. That said, you can use parentheses to be explicit about the scope of the operator’s application.

1. sum = try someThrowingFunction() + anotherThrowingFunction() // try applies to both function calls
2. sum = try (someThrowingFunction() + anotherThrowingFunction()) // try applies to both function calls
3. sum = (try someThrowingFunction()) + anotherThrowingFunction() // Error: try applies only to the first function call

A try expression can’t appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the try expression is enclosed in parentheses.

### **Literal Expression**

A literal expression consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, a playground literal, or one of the following special literals:

| **Literal** | **Type** | **Value** |
| --- | --- | --- |
| #file | String | The name of the file in which it appears. |
| #line | Int | The line number on which it appears. |
| #column | Int | The column number in which it begins. |
| #function | String | The name of the declaration in which it appears. |
| #dsohandle | UnsafeRawPointer | The DSO (dynamic shared object) handle in use where it appears. |

Inside a function, the value of #function is the name of that function, inside a method it is the name of that method, inside a property getter or setter it is the name of that property, inside special members like init or subscript it is the name of that keyword, and at the top level of a file it is the name of the current module.

When used as the default value of a function or method parameter, the special literal’s value is determined when the default value expression is evaluated at the call site.

1. func logFunctionName(string: String = #function) {
2. print(string)
3. }
4. func myFunction() {
5. logFunctionName() // Prints "myFunction()".
6. }

An array literal is an ordered collection of values. It has the following form:

1. [value 1, value 2, ...]

The last expression in the array can be followed by an optional comma. The value of an array literal has type [T], where T is the type of the expressions inside it. If there are expressions of multiple types, T is their closest common supertype. Empty array literals are written using an empty pair of square brackets and can be used to create an empty array of a specified type.

1. var emptyArray: [Double] = []

A dictionary literal is an unordered collection of key-value pairs. It has the following form:

1. [key 1: value 1, key 2: value 2, ...]

The last expression in the dictionary can be followed by an optional comma. The value of a dictionary literal has type [Key: Value], where Key is the type of its key expressions and Value is the type of its value expressions. If there are expressions of multiple types, Key and Value are the closest common supertype for their respective values. An empty dictionary literal is written as a colon inside a pair of brackets ([:]) to distinguish it from an empty array literal. You can use an empty dictionary literal to create an empty dictionary literal of specified key and value types.

1. var emptyDictionary: [String: Double] = [:]

A playground literal is used by Xcode to create an interactive representation of a color, file, or image within the program editor. Playground literals in plain text outside of Xcode are represented using a special literal syntax.

# **Statements**

In Swift, there are three kinds of statements: simple statements, compiler control statements, and control flow statements. Simple statements are the most common and consist of either an expression or a declaration. Compiler control statements allow the program to change aspects of the compiler’s behavior and include a conditional compilation block and a line control statement.

Control flow statements are used to control the flow of execution in a program. There are several types of control flow statements in Swift, including loop statements, branch statements, and control transfer statements. Loop statements allow a block of code to be executed repeatedly, branch statements allow a certain block of code to be executed only when certain conditions are met, and control transfer statements provide a way to alter the order in which code is executed. In addition, Swift provides a do statement to introduce scope, and catch and handle errors, and a defer statement for running cleanup actions just before the current scope exits.

A semicolon (;) can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

## Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has three loop statements: a for-in statement, a while statement, and a repeat-while statement.

Control flow in a loop statement can be changed by a break statement and a continue statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) and [Continue Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID442) below.

GRAMMAR OF A LOOP STATEMENT

*loop-statement* → [for-in-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_for-in-statement)

*loop-statement* → [while-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_while-statement)

*loop-statement* → [repeat-while-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_repeat-while-statement)

### **For-In Statement**

A for-in statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the [Sequence](https://developer.apple.com/documentation/swift/sequence) protocol.

A for-in statement has the following form:

1. for item in collection {
2. statements
3. }

The makeIterator() method is called on the collection expression to obtain a value of an iterator type—that is, a type that conforms to the [IteratorProtocol](https://developer.apple.com/documentation/swift/iteratorprotocol) protocol. The program begins executing a loop by calling the next() method on the iterator. If the value returned is not nil, it is assigned to the item pattern, the program executes the statements, and then continues execution at the beginning of the loop. Otherwise, the program does not perform assignment or execute the statements, and it is finished executing the for-in statement.

GRAMMAR OF A FOR-IN STATEMENT

*for-in-statement* → **for** **case***opt* [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) **in** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

### **While Statement**

A while statement allows a block of code to be executed repeatedly, as long as a condition remains true.

A while statement has the following form:

1. while condition {
2. statements
3. }

A while statement is executed as follows:

1. The condition is evaluated.

If true, execution continues to step 2. If false, the program is finished executing the while statement.

1. The program executes the statements, and execution returns to step 1.

Because the value of the condition is evaluated before the statements are executed, the statements in a while statement can be executed zero or more times.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF A WHILE STATEMENT

*while-statement* → **while** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

*condition-list* → [condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition) | [condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition) **,** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list)

*condition* → [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) | [availability-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_availability-condition) | [case-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-condition) | [optional-binding-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_optional-binding-condition)

*case-condition* → **case** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer)

*optional-binding-condition* → **let** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer) | **var** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [initializer](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_initializer)

### **Repeat-While Statement**

A repeat-while statement allows a block of code to be executed one or more times, as long as a condition remains true.

A repeat-while statement has the following form:

1. repeat {
2. statements
3. } while condition

A repeat-while statement is executed as follows:

1. The program executes the statements, and execution continues to step 2.
2. The condition is evaluated.

If true, execution returns to step 1. If false, the program is finished executing the repeat-while statement.

Because the value of the condition is evaluated after the statements are executed, the statements in a repeat-while statement are executed at least once.

The value of the condition must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF A REPEAT-WHILE STATEMENT

*repeat-while-statement* → **repeat** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) **while** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

## Branch Statements

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements: an if statement, a guard statement, and a switch statement.

Control flow in an if statement or a switch statement can be changed by a break statement and is discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) below.

GRAMMAR OF A BRANCH STATEMENT

*branch-statement* → [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

*branch-statement* → [guard-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_guard-statement)

*branch-statement* → [switch-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-statement)

### **If Statement**

An if statement is used for executing code based on the evaluation of one or more conditions.

There are two basic forms of an if statement. In each form, the opening and closing braces are required.

The first form allows code to be executed only when a condition is true and has the following form:

1. if condition {
2. statements
3. }

The second form of an if statement provides an additional else clause (introduced by the else keyword) and is used for executing one part of code when the condition is true and another part of code when the same condition is false. When a single else clause is present, an if statement has the following form:

1. if condition {
2. statements to execute if condition is true
3. } else {
4. statements to execute if condition is false
5. }

The else clause of an if statement can contain another if statement to test more than one condition. An if statement chained together in this way has the following form:

1. if condition 1 {
2. statements to execute if condition 1 is true
3. } else if condition 2 {
4. statements to execute if condition 2 is true
5. } else {
6. statements to execute if both conditions are false
7. }

The value of any condition in an if statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

GRAMMAR OF AN IF STATEMENT

*if-statement* → **if** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) [else-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-clause) *opt*

*else-clause* → **else** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) | **else** [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

### **Guard Statement**

A guard statement is used to transfer program control out of a scope if one or more conditions aren’t met.

A guard statement has the following form:

1. guard condition else {
2. statements
3. }

The value of any condition in a guard statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).

Any constants or variables assigned a value from an optional binding declaration in a guard statement condition can be used for the rest of the guard statement’s enclosing scope.

The else clause of a guard statement is required, and must either call a function with the Never return type or transfer program control outside the guard statement’s enclosing scope using one of the following statements:

* return
* break
* continue
* throw

Control transfer statements are discussed in [Control Transfer Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID440) below. For more information on functions with the Never return type, see [Functions that Never Return](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID551).

GRAMMAR OF A GUARD STATEMENT

*guard-statement* → **guard** [condition-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_condition-list) **else** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

### **Switch Statement**

A switch statement allows certain blocks of code to be executed depending on the value of a control expression.

A switch statement has the following form:

1. switch control expression {
2. case pattern 1:
3. statements
4. case pattern 2 where condition:
5. statements
6. case pattern 3 where condition,
7. pattern 4 where condition:
8. statements
9. default:
10. statements
11. }

The control expression of the switch statement is evaluated and then compared with the patterns specified in each case. If a match is found, the program executes the statements listed within the scope of that case. The scope of each case can’t be empty. As a result, you must include at least one statement following the colon (:) of each case label. Use a single break statement if you don’t intend to execute any code in the body of a matched case.

The values of expressions your code can branch on are very flexible. For example, in addition to the values of scalar types, such as integers and characters, your code can branch on the values of any type, including floating-point numbers, strings, tuples, instances of custom classes, and optionals. The value of the control expression can even be matched to the value of a case in an enumeration and checked for inclusion in a specified range of values. For examples of how to use these various types of values in switch statements, see [Switch](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID129) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

A switch case can optionally contain a where clause after each pattern. A where clause is introduced by the where keyword followed by an expression, and is used to provide an additional condition before a pattern in a case is considered matched to the control expression. If a where clause is present, the statements within the relevant case are executed only if the value of the control expression matches one of the patterns of the case and the expression of the where clause evaluates to true. For example, a control expression matches the case in the example below only if it is a tuple that contains two elements of the same value, such as (1, 1).

1. case let (x, y) where x == y:

As the above example shows, patterns in a case can also bind constants using the let keyword (they can also bind variables using the var keyword). These constants (or variables) can then be referenced in a corresponding where clause and throughout the rest of the code within the scope of the case. If the case contains multiple patterns that match the control expression, all of the patterns must contain the same constant or variable bindings, and each bound variable or constant must have the same type in all of the case’s patterns.

A switch statement can also include a default case, introduced by the default keyword. The code within a default case is executed only if no other cases match the control expression. A switch statement can include only one default case, which must appear at the end of the switch statement.

Although the actual execution order of pattern-matching operations, and in particular the evaluation order of patterns in cases, is unspecified, pattern matching in a switch statement behaves as if the evaluation is performed in source order—that is, the order in which they appear in source code. As a result, if multiple cases contain patterns that evaluate to the same value, and thus can match the value of the control expression, the program executes only the code within the first matching case in source order.

#### **Switch Statements Must Be Exhaustive**

In Swift, every possible value of the control expression’s type must match the value of at least one pattern of a case. When this simply isn’t feasible (for example, when the control expression’s type is Int), you can include a default case to satisfy the requirement.

#### **Switching Over Future Enumeration Cases**

A nonfrozen enumeration is a special kind of enumeration that may gain new enumeration cases in the future—even after you compile and ship an app. Switching over a nonfrozen enumeration requires extra consideration. When a library’s authors mark an enumeration as nonfrozen, they reserve the right to add new enumeration cases, and any code that interacts with that enumeration must be able to handle those future cases without being recompiled. Code that’s compiled in library evolution mode, code in the standard library, Swift overlays for Apple frameworks, and C and Objective-C code can declare nonfrozen enumerations. For information about frozen and nonfrozen enumerations, see [frozen](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID620).

When switching over a nonfrozen enumeration value, you always need to include a default case, even if every case of the enumeration already has a corresponding switch case. You can apply the @unknown attribute to the default case, which indicates that the default case should match only enumeration cases that are added in the future. Swift produces a warning if the default case matches any enumeration case that is known at compiler time. This future warning informs you that the library author added a new case to the enumeration that doesn’t have a corresponding switch case.

The following example switches over all three existing cases of the standard library’s [Mirror.AncestorRepresentation](https://developer.apple.com/documentation/swift/mirror/ancestorrepresentation) enumeration. If you add additional cases in the future, the compiler generates a warning to indicate that you need to update the switch statement to take the new cases into account.

1. let representation: Mirror.AncestorRepresentation = .generated
2. switch representation {
3. case .customized:
4. print("Use the nearest ancestor’s implementation.")
5. case .generated:
6. print("Generate a default mirror for all ancestor classes.")
7. case .suppressed:
8. print("Suppress the representation of all ancestor classes.")
9. @unknown default:
10. print("Use a representation that was unknown when this code was compiled.")
11. }
12. // Prints "Generate a default mirror for all ancestor classes."

#### **Execution Does Not Fall Through Cases Implicitly**

After the code within a matched case has finished executing, the program exits from the switch statement. Program execution does not continue or “fall through” to the next case or default case. That said, if you want execution to continue from one case to the next, explicitly include a fallthrough statement, which simply consists of the fallthrough keyword, in the case from which you want execution to continue. For more information about the fallthrough statement, see [Fallthrough Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html" \l "ID443) below.

GRAMMAR OF A SWITCH STATEMENT

*switch-statement* → **switch** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) **{** [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt* **}**

*switch-cases* → [switch-case](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-case) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-case* → [case-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-label) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements)

*switch-case* → [default-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_default-label) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements)

*switch-case* → [conditional-switch-case](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_conditional-switch-case)

*case-label* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **case** [case-item-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-item-list) **:**

*case-item-list* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* | [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* **,** [case-item-list](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_case-item-list)

*default-label* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **default** **:**

*where-clause* → **where** [where-expression](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-expression)

*where-expression* → [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

*conditional-switch-case* → [switch-if-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-if-directive-clause) [switch-elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-elseif-directive-clauses) *opt* [switch-else-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-else-directive-clause) *opt* [endif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_endif-directive)

*switch-if-directive-clause* → [if-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-elseif-directive-clauses* → [elseif-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clause) [switch-elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-elseif-directive-clauses) *opt*

*switch-elseif-directive-clause* → [elseif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

*switch-else-directive-clause* → [else-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive) [switch-cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-cases) *opt*

## Labeled Statement

You can prefix a loop statement, an if statement, a switch statement, or a do statement with a statement label, which consists of the name of the label followed immediately by a colon (:). Use statement labels with break and continue statements to be explicit about how you want to change control flow in a loop statement or a switch statement, as discussed in [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) and [Continue Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID442) below.

The scope of a labeled statement is the entire statement following the statement label. You can nest labeled statements, but the name of each statement label must be unique.

For more information and to see examples of how to use statement labels, see [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A LABELED STATEMENT

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [loop-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_loop-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [if-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [switch-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_switch-statement)

*labeled-statement* → [statement-label](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statement-label) [do-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_do-statement)

*statement-label* → [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) **:**

*label-name* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

## Control Transfer Statements

Control transfer statements can change the order in which code in your program is executed by unconditionally transferring program control from one piece of code to another. Swift has five control transfer statements: a break statement, a continue statement, a fallthrough statement, a return statement, and a throw statement.

GRAMMAR OF A CONTROL TRANSFER STATEMENT

*control-transfer-statement* → [break-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_break-statement)

*control-transfer-statement* → [continue-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_continue-statement)

*control-transfer-statement* → [fallthrough-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html" \l "grammar_fallthrough-statement)

*control-transfer-statement* → [return-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_return-statement)

*control-transfer-statement* → [throw-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_throw-statement)

### **Break Statement**

A break statement ends program execution of a loop, an if statement, or a switch statement. A break statement can consist of only the break keyword, or it can consist of the break keyword followed by the name of a statement label, as shown below.

1. break
2. break label name

When a break statement is followed by the name of a statement label, it ends program execution of the loop, if statement, or switch statement named by that label.

When a break statement is not followed by the name of a statement label, it ends program execution of the switch statement or the innermost enclosing loop statement in which it occurs. You can’t use an unlabeled break statement to break out of an if statement.

In both cases, program control is then transferred to the first line of code following the enclosing loop or switch statement, if any.

For examples of how to use a break statement, see [Break](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID137) and [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A BREAK STATEMENT

*break-statement* → **break** [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) *opt*

### **Continue Statement**

A continue statement ends program execution of the current iteration of a loop statement but does not stop execution of the loop statement. A continue statement can consist of only the continue keyword, or it can consist of the continue keyword followed by the name of a statement label, as shown below.

1. continue
2. continue label name

When a continue statement is followed by the name of a statement label, it ends program execution of the current iteration of the loop statement named by that label.

When a continue statement is not followed by the name of a statement label, it ends program execution of the current iteration of the innermost enclosing loop statement in which it occurs.

In both cases, program control is then transferred to the condition of the enclosing loop statement.

In a for statement, the increment expression is still evaluated after the continue statement is executed, because the increment expression is evaluated after the execution of the loop’s body.

For examples of how to use a continue statement, see [Continue](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID136) and [Labeled Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID141) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A CONTINUE STATEMENT

*continue-statement* → **continue** [label-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_label-name) *opt*

### **Fallthrough Statement**

A fallthrough statement consists of the fallthrough keyword and occurs only in a case block of a switch statement. A fallthrough statement causes program execution to continue from one case in a switch statement to the next case. Program execution continues to the next case even if the patterns of the case label do not match the value of the switch statement’s control expression.

A fallthrough statement can appear anywhere inside a switch statement, not just as the last statement of a case block, but it can’t be used in the final case block. It also cannot transfer control into a case block whose pattern contains value binding patterns.

For an example of how to use a fallthrough statement in a switch statement, see [Control Transfer Statements](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID135) in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html).

GRAMMAR OF A FALLTHROUGH STATEMENT

*fallthrough-statement* → **fallthrough**

### **Return Statement**

A return statement occurs in the body of a function or method definition and causes program execution to return to the calling function or method. Program execution continues at the point immediately following the function or method call.

A return statement can consist of only the return keyword, or it can consist of the return keyword followed by an expression, as shown below.

1. return
2. return expression

When a return statement is followed by an expression, the value of the expression is returned to the calling function or method. If the value of the expression does not match the value of the return type declared in the function or method declaration, the expression’s value is converted to the return type before it is returned to the calling function or method.

NOTE

As described in [Failable Initializers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html" \l "ID376), a special form of the return statement (return nil) can be used in a failable initializer to indicate initialization failure.

When a return statement is not followed by an expression, it can be used only to return from a function or method that does not return a value (that is, when the return type of the function or method is Void or ()).

GRAMMAR OF A RETURN STATEMENT

*return-statement* → **return** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression) *opt*

### **Throw Statement**

A throw statement occurs in the body of a throwing function or method, or in the body of a closure expression whose type is marked with the throws keyword.

A throw statement causes a program to end execution of the current scope and begin error propagation to its enclosing scope. The error that’s thrown continues to propagate until it’s handled by a catch clause of a do statement.

A throw statement consists of the throw keyword followed by an expression, as shown below.

1. throw expression

The value of the expression must have a type that conforms to the Error protocol.

For an example of how to use a throw statement, see [Propagating Errors Using Throwing Functions](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID510) in [Error Handling](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html).

GRAMMAR OF A THROW STATEMENT

*throw-statement* → **throw** [expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_expression)

## Defer Statement

A defer statement is used for executing code just before transferring program control outside of the scope that the defer statement appears in.

A defer statement has the following form:

1. defer {
2. statements
3. }

The statements within the defer statement are executed no matter how program control is transferred. This means that a defer statement can be used, for example, to perform manual resource management such as closing file descriptors, and to perform actions that need to happen even if an error is thrown.

If multiple defer statements appear in the same scope, the order they appear is the reverse of the order they are executed. Executing the last defer statement in a given scope first means that statements inside that last defer statement can refer to resources that will be cleaned up by other defer statements.

1. func f() {
2. defer { print("First defer") }
3. defer { print("Second defer") }
4. print("End of function")
5. }
6. f()
7. // Prints "End of function"
8. // Prints "Second defer"
9. // Prints "First defer"

The statements in the defer statement can’t transfer program control outside of the defer statement.

GRAMMAR OF A DEFER STATEMENT

*defer-statement* → **defer** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

## Do Statement

The do statement is used to introduce a new scope and can optionally contain one or more catch clauses, which contain patterns that match against defined error conditions. Variables and constants declared in the scope of a do statement can be accessed only within that scope.

A do statement in Swift is similar to curly braces ({}) in C used to delimit a code block, and does not incur a performance cost at runtime.

A do statement has the following form:

1. do {
2. try expression
3. statements
4. } catch pattern 1 {
5. statements
6. } catch pattern 2 where condition {
7. statements
8. }

Like a switch statement, the compiler attempts to infer whether catch clauses are exhaustive. If such a determination can be made, the error is considered handled. Otherwise, the error can propagate out of the containing scope, which means the error must be handled by an enclosing catch clause or the containing function must be declared with throws.

To ensure that an error is handled, use a catch clause with a pattern that matches all errors, such as a wildcard pattern (\_). If a catch clause does not specify a pattern, the catch clause matches and binds any error to a local constant named error. For more information about the patterns you can use in a catch clause, see [Patterns](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html).

To see an example of how to use a do statement with several catch clauses, see [Handling Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID512).

GRAMMAR OF A DO STATEMENT

*do-statement* → **do** [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block) [catch-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clauses) *opt*

*catch-clauses* → [catch-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clause) [catch-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_catch-clauses) *opt*

*catch-clause* → **catch** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) *opt* [where-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_where-clause) *opt* [code-block](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_code-block)

## Compiler Control Statements

Compiler control statements allow the program to change aspects of the compiler’s behavior. Swift has three compiler control statements: a conditional compilation block a line control statement, and a compile-time diagnostic statement.

GRAMMAR OF A COMPILER CONTROL STATEMENT

*compiler-control-statement* → [conditional-compilation-block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_conditional-compilation-block)

*compiler-control-statement* → [line-control-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_line-control-statement)

*compiler-control-statement* → [diagnostic-statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-statement)

### **Conditional Compilation Block**

A conditional compilation block allows code to be conditionally compiled depending on the value of one or more compilation conditions.

Every conditional compilation block begins with the #if compilation directive and ends with the #endif compilation directive. A simple conditional compilation block has the following form:

1. #if compilation condition
2. statements
3. #endif

Unlike the condition of an if statement, the compilation condition is evaluated at compile time. As a result, the statements are compiled and executed only if the compilation condition evaluates to true at compile time.

The compilation condition can include the true and false Boolean literals, an identifier used with the -D command line flag, or any of the platform conditions listed in the table below.

| **Platform condition** | **Valid arguments** |
| --- | --- |
| os() | macOS, iOS, watchOS, tvOS, Linux |
| arch() | i386, x86\_64, arm, arm64 |
| swift() | >= or < followed by a version number |
| compiler() | >= or < followed by a version number |
| canImport() | A module name |
| targetEnvironment() | simulator, macCatalyst |

The version number for the swift() and compiler() platform conditions consists of a major number, optional minor number, optional patch number, and so on, with a dot (.) separating each part of the version number. There must not be whitespace between the comparison operator and the version number. The version for compiler() is the compiler version, regardless of the Swift version setting passed to the compiler. The version for swift() is the language version currently being compiled. For example, if you compile your code using the Swift 5 compiler in Swift 4.2 mode, the compiler version is 5 and the language version is 4.2. With those settings, the following code prints all three messages:

1. #if compiler(>=5)
2. print("Compiled with the Swift 5 compiler or later")
3. #endif
4. #if swift(>=4.2)
5. print("Compiled in Swift 4.2 mode or later")
6. #endif
7. #if compiler(>=5) && swift(<5)
8. print("Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5")
9. #endif
10. // Prints "Compiled with the Swift 5 compiler or later"
11. // Prints "Compiled in Swift 4.2 mode or later"
12. // Prints "Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5"

The argument for the canImport() platform condition is the name of a module that may not be present on all platforms. This condition tests whether it’s possible to import the module, but doesn’t actually import it. If the module is present, the platform condition returns true; otherwise, it returns false.

The targetEnvironment() platform condition returns true when code is compiled for a simulator; otherwise, it returns false.

NOTE

The arch(arm) platform condition does not return true for ARM 64 devices. The arch(i386) platform condition returns true when code is compiled for the 32–bit iOS simulator.

You can combine compilation conditions using the logical operators &&, ||, and ! and use parentheses for grouping. These operators have the same associativity and precedence as the logical operators that are used to combine ordinary Boolean expressions.

Similar to an if statement, you can add multiple conditional branches to test for different compilation conditions. You can add any number of additional branches using #elseif clauses. You can also add a final additional branch using an #else clause. Conditional compilation blocks that contain multiple branches have the following form:

1. #if compilation condition 1
2. statements to compile if compilation condition 1 is true
3. #elseif compilation condition 2
4. statements to compile if compilation condition 2 is true
5. #else
6. statements to compile if both compilation conditions are false
7. #endif

NOTE

Each statement in the body of a conditional compilation block is parsed even if it’s not compiled. However, there is an exception if the compilation condition includes a swift() platform condition: The statements are parsed only if the compiler’s version of Swift matches what is specified in the platform condition. This exception ensures that an older compiler doesn’t attempt to parse syntax introduced in a newer version of Swift.

GRAMMAR OF A CONDITIONAL COMPILATION BLOCK

*conditional-compilation-block* → [if-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive-clause) [elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clauses) *opt* [else-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive-clause) *opt* [endif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_endif-directive)

*if-directive-clause* → [if-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_if-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*elseif-directive-clauses* → [elseif-directive-clause](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clause) [elseif-directive-clauses](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive-clauses) *opt*

*elseif-directive-clause* → [elseif-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_elseif-directive) [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*else-directive-clause* → [else-directive](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_else-directive) [statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_statements) *opt*

*if-directive* → **#if**

*elseif-directive* → **#elseif**

*else-directive* → **#else**

*endif-directive* → **#endif**

*compilation-condition* → [platform-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_platform-condition)

*compilation-condition* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

*compilation-condition* → [boolean-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html" \l "grammar_boolean-literal)

*compilation-condition* → **(** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **)**

*compilation-condition* → **!** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*compilation-condition* → [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **&&** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*compilation-condition* → [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition) **||** [compilation-condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_compilation-condition)

*platform-condition* → **os** **(** [operating-system](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_operating-system) **)**

*platform-condition* → **arch** **(** [architecture](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_architecture) **)**

*platform-condition* → **swift** **(** **>=** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)** | **swift** **(** **<** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)**

*platform-condition* → **compiler** **(** **>=** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)** | **compiler** **(** **<** [swift-version](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version) **)**

*platform-condition* → **canImport** **(** [module-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_module-name) **)**

*platform-condition* → **targetEnvironment** **(** [environment](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_environment) **)**

*operating-system* → **macOS** | **iOS** | **watchOS** | **tvOS**

*architecture* → **i386** | **x86\_64** | **arm** | **arm64**

*swift-version* → [decimal-digits](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_decimal-digits) [swift-version-continuation](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version-continuation) *opt*

*swift-version-continuation* → **.** [decimal-digits](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_decimal-digits) [swift-version-continuation](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_swift-version-continuation) *opt*

*module-name* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

*environment* → **simulator**

### **Line Control Statement**

A line control statement is used to specify a line number and filename that can be different from the line number and filename of the source code being compiled. Use a line control statement to change the source code location used by Swift for diagnostic and debugging purposes.

A line control statement has the following forms:

1. #sourceLocation(file: filename, line: line number)
2. #sourceLocation()

The first form of a line control statement changes the values of the #line and #file literal expressions, beginning with the line of code following the line control statement. The line number changes the value of #line and is any integer literal greater than zero. The filename changes the value of #file and is a string literal.

The second form of a line control statement, #sourceLocation(), resets the source code location back to the default line numbering and filename.

GRAMMAR OF A LINE CONTROL STATEMENT

*line-control-statement* → **#sourceLocation** **(** **file:** [file-name](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_file-name) **,** **line:** [line-number](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_line-number) **)**

*line-control-statement* → **#sourceLocation** **(** **)**

*line-number* → A decimal integer greater than zero

*file-name* → [static-string-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_static-string-literal)

### **Compile-Time Diagnostic Statement**

A compile-time diagnostic statement causes the compiler to emit an error or a warning during compilation. A compile-time diagnostic statement has the following forms:

1. #error("error message")
2. #warning("warning message")

The first form emits the error message as a fatal error and terminates the compilation process. The second form emits the warning message as a nonfatal warning and allows compilation to proceed. You write the diagnostic message as a static string literal. Static string literals can’t use features like string interpolation or concatenation, but they can use the multiline string literal syntax.

GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT

*diagnostic-statement* → **#error** **(** [diagnostic-message](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-message) **)**

*diagnostic-statement* → **#warning** **(** [diagnostic-message](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#grammar_diagnostic-message) **)**

*diagnostic-message* → [static-string-literal](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_static-string-literal)

## Availability Condition

An availability condition is used as a condition of an if, while, and guard statement to query the availability of APIs at runtime, based on specified platforms arguments.

An availability condition has the following form:

1. if #available(platform name version, ..., \*) {
2. statements to execute if the APIs are available
3. } else {
4. fallback statements to execute if the APIs are unavailable
5. }

You use an availability condition to execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

The availability condition takes a comma-separated list of platform names and versions. Use iOS, macOS, watchOS, and tvOS for the platform names, and include the corresponding version numbers. The \* argument is required and specifies that on any other platform, the body of the code block guarded by the availability condition executes on the minimum deployment target specified by your target.

### **Throwing Functions and Methods**

Functions and methods that can throw an error must be marked with the throws keyword. These functions and methods are known as throwing functions and throwing methods. They have the following form:

1. func function name(parameters) throws -> return type {
2. statements
3. }

Calls to a throwing function or method must be wrapped in a try or try! expression (that is, in the scope of a try or try! operator).

The throws keyword is part of a function’s type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one.

You can’t overload a function based only on whether the function can throw an error. That said, you can overload a function based on whether a function parameter can throw an error.

A throwing method can’t override a nonthrowing method, and a throwing method can’t satisfy a protocol requirement for a nonthrowing method. That said, a nonthrowing method can override a throwing method, and a nonthrowing method can satisfy a protocol requirement for a throwing method.

### **Rethrowing Functions and Methods**

A function or method can be declared with the rethrows keyword to indicate that it throws an error only if one of its function parameters throws an error. These functions and methods are known as rethrowing functions and rethrowing methods. Rethrowing functions and methods must have at least one throwing function parameter.

1. func someFunction(callback: () throws -> Void) rethrows {
2. try callback()
3. }

A rethrowing function or method can contain a throw statement only inside a catch clause. This lets you call the throwing function inside a do-catch block and handle errors in the catch clause by throwing a different error. In addition, the catch clause must handle only errors thrown by one of the rethrowing function’s throwing parameters. For example, the following is invalid because the catch clause would handle the error thrown by alwaysThrows().

1. func alwaysThrows() throws {
2. throw SomeError.error
3. }
4. func someFunction(callback: () throws -> Void) rethrows {
5. do {
6. try callback()
7. try alwaysThrows() // Invalid, alwaysThrows() isn't a throwing parameter
8. } catch {
9. throw AnotherError.error
10. }
11. }

A throwing method can’t override a rethrowing method, and a throwing method can’t satisfy a protocol requirement for a rethrowing method. That said, a rethrowing method can override a throwing method, and a rethrowing method can satisfy a protocol requirement for a throwing method.

# **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:

1. @attribute name
2. @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:
  1. 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:
  1. 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:
  1. 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:
  1. 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:
  1. 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
  2. protocol MyProtocol {
  3. // protocol definition
  4. }
  5. // Subsequent release renames MyProtocol
  6. protocol MyRenamedProtocol {
  7. // protocol definition
  8. }
  9. @available(\*, unavailable, renamed: "MyRenamedProtocol")
  10. 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:

1. @available(platform name version number, \*)
2. @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.

1. @available(iOS 10.0, macOS 10.12, \*)
2. class MyClass {
3. // class definition
4. }

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.

1. @available(swift 3.0.2)
2. @available(macOS 10.12, \*)
3. struct MyStruct {
4. // struct definition
5. }

### **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.

1. @dynamicCallable
2. struct TelephoneExchange {
3. func dynamicallyCall(withArguments phoneNumber: [Int]) {
4. if phoneNumber == [4, 1, 1] {
5. print("Get Swift help on forums.swift.org")
6. } else {
7. print("Unrecognized number")
8. }
9. }
10. }
11. let dial = TelephoneExchange()
12. // Use a dynamic method call.
13. dial(4, 1, 1)
14. // Prints "Get Swift help on forums.swift.org"
15. dial(8, 6, 7, 5, 3, 0, 9)
16. // Prints "Unrecognized number"
17. // Call the underlying method directly.
18. dial.dynamicallyCall(withArguments: [4, 1, 1])

The declaration of the dynamicallyCall(withArguments:) method must have a single parameter that conforms to the [ExpressibleByArrayLiteral](https://developer.apple.com/documentation/swift/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.

1. @dynamicCallable
2. struct Repeater {
3. func dynamicallyCall(withKeywordArguments pairs: KeyValuePairs<String, Int>) -> String {
4. return pairs
5. .map { label, count in
6. repeatElement(label, count: count).joined(separator: " ")
7. }
8. .joined(separator: "\n")
9. }
10. }
11. let repeatLabels = Repeater()
12. print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
13. // a
14. // b b
15. // c c c
16. // b b
17. // a

The declaration of the dynamicallyCall(withKeywordArguments:) method must have a single parameter that conforms to the [ExpressibleByDictionaryLiteral](https://developer.apple.com/documentation/swift/expressiblebydictionaryliteral) protocol, and the return type can be any type. The parameter’s [Key](https://developer.apple.com/documentation/swift/expressiblebydictionaryliteral/2294108-key) must be [ExpressibleByStringLiteral](https://developer.apple.com/documentation/swift/expressiblebystringliteral). The previous example uses [KeyValuePairs](https://developer.apple.com/documentation/swift/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>.

1. 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](https://developer.apple.com/documentation/swift/keypath), [WritableKeyPath](https://developer.apple.com/documentation/swift/writablekeypath), or [ReferenceWritableKeyPath](https://developer.apple.com/documentation/swift/referencewritablekeypath). It can accept member names using an argument of a type that conforms to the [ExpressibleByStringLiteral](https://developer.apple.com/documentation/swift/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:

1. @dynamicMemberLookup
2. struct DynamicStruct {
3. let dictionary = ["someDynamicMember": 325,
4. "someOtherMember": 787]
5. subscript(dynamicMember member: String) -> Int {
6. return dictionary[member] ?? 1054
7. }
8. }
9. let s = DynamicStruct()
10. // Use dynamic member lookup.
11. let dynamic = s.someDynamicMember
12. print(dynamic)
13. // Prints "325"
14. // Call the underlying subscript directly.
15. let equivalent = s[dynamicMember: "someDynamicMember"]
16. print(dynamic == equivalent)
17. // 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:

1. struct Point { var x, y: Int }
2. @dynamicMemberLookup
3. struct PassthroughWrapper<Value> {
4. var value: Value
5. subscript<T>(dynamicMember member: KeyPath<Value, T>) -> T {
6. get { return value[keyPath: member] }
7. }
8. }
9. let point = Point(x: 381, y: 431)
10. let wrapper = PassthroughWrapper(value: point)
11. print(wrapper.x)

### **frozen**

Apply this attribute to a structure or enumeration declaration to restrict the kinds of changes you can make to the type. This attribute is allowed only when compiling in library evolution mode. Future versions of the library can’t change the declaration by adding, removing, or reordering an enumeration’s cases or a structure’s stored instance properties. These changes are allowed on nonfrozen types, but they break ABI compatibility for frozen types.

NOTE

When the compiler isn’t in library evolution mode, all structures and enumerations are implicitly frozen, and you can’t use this attribute.

In library evolution mode, code that interacts with members of nonfrozen structures and enumerations is compiled in a way that allows it to continue working without recompiling even if a future version of the library adds, removes, or reorders some of that type’s members. The compiler makes this possible using techniques like looking up information at runtime and adding a layer of indirection. Marking a structure or enumeration as frozen gives up this flexibility to gain performance: Future versions of the library can make only limited changes to the type, but the compiler can make additional optimizations in code that interacts with the type’s members.

Frozen types, the types of the stored properties of frozen structures, and the associated values of frozen enumeration cases must be public or marked with the usableFromInline attribute. The properties of a frozen structure can’t have property observers, and expressions that provide the initial value for stored instance properties must follow the same restrictions as inlinable functions, as discussed in [inlinable](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID587).

To enable library evolution mode on the command line, pass the -enable-library-evolution option to the Swift compiler. To enable it in Xcode, set the “Build Libraries for Distribution” build setting (BUILD\_LIBRARY\_FOR\_DISTRIBUTION) to Yes, as described in [Xcode Help](https://help.apple.com/xcode/mac/current/" \l "/dev04b3a04ba).

A switch statement over a frozen enumeration doesn’t require a default case, as discussed in [Switching Over Future Enumeration Cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602). Including a default or @unknown default case when switching over a frozen enumeration produces a warning because that code is never executed.

### **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:

1. import AppKit
2. 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](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html#//apple_ref/doc/uid/TP40011210-CH10-SW1) in [Programming with Objective-C](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011210). 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.

1. class ExampleClass: NSObject {
2. @objc var enabled: Bool {
3. @objc(isEnabled) get {
4. // Return the appropriate value
5. }
6. }
7. }

### **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.

### **propertyWrapper**

Apply this attribute to a class, structure, or enumeration declaration to use that type as a property wrapper. When you apply this attribute to a type, you create a custom attribute with the same name as the type. Apply that new attribute to a property of a class, structure, or enumeration to wrap access to the property through an instance of the wrapper type. Local and global variables can’t use property wrappers.

The wrapper must define a wrappedValue instance property. The wrapped value of the property is the value that the getter and setter for this property expose. In most cases, wrappedValue is a computed value, but it can be a stored value instead. The wrapper is responsible for defining and managing any underlying storage needed by its wrapped value. The compiler synthesizes storage for the instance of the wrapper type by prefixing the name of the wrapped property with an underscore (\_)—for example, the wrapper for someProperty is stored as \_someProperty. The synthesized storage for the wrapper has an access control level of private.

A property that has a property wrapper can include willSet and didSet blocks, but it can’t override the compiler-synthesized get or set blocks.

Swift provides two forms of syntactic sugar for initialization of a property wrapper. You can use assignment syntax in the definition of a wrapped value to pass the expression on the right-hand side of the assignment as the argument to the wrappedValue parameter of the property wrapper’s initializer. You can also provide arguments to the attribute when you apply it to a property, and those arguments are passed to the property wrapper’s initializer. For example, in the code below, SomeStruct calls each of the initializers that SomeWrapper defines.

1. @propertyWrapper
2. struct SomeWrapper {
3. var wrappedValue: Int
4. var someValue: Double
5. init() {
6. self.wrappedValue = 100
7. self.someValue = 12.3
8. }
9. init(wrappedValue: Int) {
10. self.wrappedValue = wrappedValue
11. self.someValue = 45.6
12. }
13. init(wrappedValue value: Int, custom: Double) {
14. self.wrappedValue = value
15. self.someValue = custom
16. }
17. }
18. struct SomeStruct {
19. // Uses init()
20. @SomeWrapper var a: Int
21. // Uses init(wrappedValue:)
22. @SomeWrapper var b = 10
23. // Both use init(wrappedValue:custom:)
24. @SomeWrapper(custom: 98.7) var c = 30
25. @SomeWrapper(wrappedValue: 30, custom: 98.7) var d
26. }

The projected value for a wrapped property is a second value that a property wrapper can use to expose additional functionality. The author of a property wrapper type is responsible for determining the meaning of its projected value and defining the interface that the projected value exposes. To project a value from a property wrapper, define a projectedValue instance property on the wrapper type. The compiler synthesizes an identifier for the projected value by prefixing the name of the wrapped property with a dollar sign ($)—for example, the projected value for someProperty is $someProperty. The projected value has the same access control level as the original wrapped property.

1. @propertyWrapper
2. struct WrapperWithProjection {
3. var wrappedValue: Int
4. var projectedValue: SomeProjection {
5. return SomeProjection(wrapper: self)
6. }
7. }
8. struct SomeProjection {
9. var wrapper: WrapperWithProjection
10. }
11. struct SomeStruct {
12. @WrapperWithProjection var x = 123
13. }
14. let s = SomeStruct()
15. s.x // Int value
16. s.$x // SomeProjection value
17. s.$x.wrapper // WrapperWithProjection value

### **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(\_:\_:\_:\_:)](https://developer.apple.com/documentation/uikit/1622933-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. A structure or class marked usableFromInline can use only types that are public or usableFromInline for its properties. An enumeration marked usableFromInline can use only types that are public or usableFromInline for the raw values and associated values of its cases.

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(\_:\_:)](https://developer.apple.com/documentation/swift/1538339-min/) function and a [min()](https://developer.apple.com/documentation/swift/sequence/1641174-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](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) and [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449).

### **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](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546).

## 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](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602).

GRAMMAR OF AN ATTRIBUTE

# **Patterns**

A pattern represents the structure of a single value or a composite value. For example, the structure of a tuple (1, 2) is a comma-separated list of two elements. Because patterns represent the structure of a value rather than any one particular value, you can match them with a variety of values. For instance, the pattern (x, y) matches the tuple (1, 2) and any other two-element tuple. In addition to matching a pattern with a value, you can extract part or all of a composite value and bind each part to a constant or variable name.

In Swift, there are two basic kinds of patterns: those that successfully match any kind of value, and those that may fail to match a specified value at runtime.

The first kind of pattern is used for destructuring values in simple variable, constant, and optional bindings. These include wildcard patterns, identifier patterns, and any value binding or tuple patterns containing them. You can specify a type annotation for these patterns to constrain them to match only values of a certain type.

The second kind of pattern is used for full pattern matching, where the values you’re trying to match against may not be there at runtime. These include enumeration case patterns, optional patterns, expression patterns, and type-casting patterns. You use these patterns in a case label of a switch statement, a catch clause of a do statement, or in the case condition of an if, while, guard, or for-in statement.

GRAMMAR OF A PATTERN

*pattern* → [wildcard-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_wildcard-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [identifier-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_identifier-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [value-binding-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_value-binding-pattern)

*pattern* → [tuple-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) *opt*

*pattern* → [enum-case-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html" \l "grammar_enum-case-pattern)

*pattern* → [optional-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_optional-pattern)

*pattern* → [type-casting-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_type-casting-pattern)

*pattern* → [expression-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_expression-pattern)

## Wildcard Pattern

A wildcard pattern matches and ignores any value and consists of an underscore (\_). Use a wildcard pattern when you don’t care about the values being matched against. For example, the following code iterates through the closed range 1...3, ignoring the current value of the range on each iteration of the loop:

1. for \_ in 1...3 {
2. // Do something three times.
3. }

GRAMMAR OF A WILDCARD PATTERN

*wildcard-pattern* → **\_**

## Identifier Pattern

An identifier pattern matches any value and binds the matched value to a variable or constant name. For example, in the following constant declaration, someValue is an identifier pattern that matches the value 42 of type Int:

1. let someValue = 42

When the match succeeds, the value 42 is bound (assigned) to the constant name someValue.

When the pattern on the left-hand side of a variable or constant declaration is an identifier pattern, the identifier pattern is implicitly a subpattern of a value-binding pattern.

GRAMMAR OF AN IDENTIFIER PATTERN

*identifier-pattern* → [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier)

## Value-Binding Pattern

A value-binding pattern binds matched values to variable or constant names. Value-binding patterns that bind a matched value to the name of a constant begin with the let keyword; those that bind to the name of variable begin with the var keyword.

Identifiers patterns within a value-binding pattern bind new named variables or constants to their matching values. For example, you can decompose the elements of a tuple and bind the value of each element to a corresponding identifier pattern.

1. let point = (3, 2)
2. switch point {
3. // Bind x and y to the elements of point.
4. case let (x, y):
5. print("The point is at (\(x), \(y)).")
6. }
7. // Prints "The point is at (3, 2)."

In the example above, let distributes to each identifier pattern in the tuple pattern (x, y). Because of this behavior, the switch cases case let (x, y): and case (let x, let y): match the same values.

GRAMMAR OF A VALUE-BINDING PATTERN

*value-binding-pattern* → **var** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) | **let** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern)

## Tuple Pattern

A tuple pattern is a comma-separated list of zero or more patterns, enclosed in parentheses. Tuple patterns match values of corresponding tuple types.

You can constrain a tuple pattern to match certain kinds of tuple types by using type annotations. For example, the tuple pattern (x, y): (Int, Int) in the constant declaration let (x, y): (Int, Int) = (1, 2) matches only tuple types in which both elements are of type Int.

When a tuple pattern is used as the pattern in a for-in statement or in a variable or constant declaration, it can contain only wildcard patterns, identifier patterns, optional patterns, or other tuple patterns that contain those. For example, the following code isn’t valid because the element 0 in the tuple pattern (x, 0) is an expression pattern:

1. let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
2. // This code isn't valid.
3. for (x, 0) in points {
4. /\* ... \*/
5. }

The parentheses around a tuple pattern that contains a single element have no effect. The pattern matches values of that single element’s type. For example, the following are equivalent:

1. let a = 2 // a: Int = 2
2. let (a) = 2 // a: Int = 2
3. let (a): Int = 2 // a: Int = 2

GRAMMAR OF A TUPLE PATTERN

*tuple-pattern* → **(** [tuple-pattern-element-list](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element-list) *opt* **)**

*tuple-pattern-element-list* → [tuple-pattern-element](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element) | [tuple-pattern-element](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element) **,** [tuple-pattern-element-list](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern-element-list)

*tuple-pattern-element* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) | [identifier](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#grammar_identifier) **:** [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern)

## Enumeration Case Pattern

An enumeration case pattern matches a case of an existing enumeration type. Enumeration case patterns appear in switch statement case labels and in the case conditions of if, while, guard, and for-in statements.

If the enumeration case you’re trying to match has any associated values, the corresponding enumeration case pattern must specify a tuple pattern that contains one element for each associated value. For an example that uses a switch statement to match enumeration cases containing associated values, see [Associated Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID148).

An enumeration case pattern also matches values of that case wrapped in an optional. This simplified syntax lets you omit an optional pattern. Note that, because Optional is implemented as an enumeration, .none and .some can appear in the same switch as the cases of the enumeration type.

1. enum SomeEnum { case left, right }
2. let x: SomeEnum? = .left
3. switch x {
4. case .left:
5. print("Turn left")
6. case .right:
7. print("Turn right")
8. case nil:
9. print("Keep going straight")
10. }
11. // Prints "Turn left"

GRAMMAR OF AN ENUMERATION CASE PATTERN

*enum-case-pattern* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) *opt* **.** [enum-case-name](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#grammar_enum-case-name) [tuple-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_tuple-pattern) *opt*

## Optional Pattern

An optional pattern matches values wrapped in a some(Wrapped) case of an Optional<Wrapped> enumeration. Optional patterns consist of an identifier pattern followed immediately by a question mark and appear in the same places as enumeration case patterns.

Because optional patterns are syntactic sugar for Optional enumeration case patterns, the following are equivalent:

1. let someOptional: Int? = 42
2. // Match using an enumeration case pattern.
3. if case .some(let x) = someOptional {
4. print(x)
5. }
6. // Match using an optional pattern.
7. if case let x? = someOptional {
8. print(x)
9. }

The optional pattern provides a convenient way to iterate over an array of optional values in a for-in statement, executing the body of the loop only for non-nil elements.

1. let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
2. // Match only non-nil values.
3. for case let number? in arrayOfOptionalInts {
4. print("Found a \(number)")
5. }
6. // Found a 2
7. // Found a 3
8. // Found a 5

GRAMMAR OF AN OPTIONAL PATTERN

*optional-pattern* → [identifier-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_identifier-pattern) **?**

## Type-Casting Patterns

There are two type-casting patterns, the is pattern and the as pattern. The is pattern appears only in switch statement case labels. The is and as patterns have the following form:

1. is type
2. pattern as type

The is pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the is pattern—or a subclass of that type. The is pattern behaves like the is operator in that they both perform a type cast but discard the returned type.

The as pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the as pattern—or a subclass of that type. If the match succeeds, the type of the matched value is cast to the pattern specified in the right-hand side of the as pattern.

For an example that uses a switch statement to match values with is and as patterns, see [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342).

GRAMMAR OF A TYPE CASTING PATTERN

*type-casting-pattern* → [is-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_is-pattern) | [as-pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_as-pattern)

*is-pattern* → **is** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

*as-pattern* → [pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#grammar_pattern) **as** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

## Expression Pattern

An expression pattern represents the value of an expression. Expression patterns appear only in switch statement case labels.

The expression represented by the expression pattern is compared with the value of an input expression using the Swift standard library ~= operator. The matches succeeds if the ~= operator returns true. By default, the ~= operator compares two values of the same type using the == operator. It can also match a value with a range of values, by checking whether the value is contained within the range, as the following example shows.

1. let point = (1, 2)
2. switch point {
3. case (0, 0):
4. print("(0, 0) is at the origin.")
5. case (-2...2, -2...2):
6. print("(\(point.0), \(point.1)) is near the origin.")
7. default:
8. print("The point is at (\(point.0), \(point.1)).")
9. }
10. // Prints "(1, 2) is near the origin."

You can overload the ~= operator to provide custom expression matching behavior. For example, you can rewrite the above example to compare the point expression with a string representations of points.

1. // Overload the ~= operator to match a string with an integer.
2. func ~= (pattern: String, value: Int) -> Bool {
3. return pattern == "\(value)"
4. }
5. switch point {
6. case ("0", "0"):
7. print("(0, 0) is at the origin.")
8. default:
9. print("The point is at (\(point.0), \(point.1)).")
10. }
11. // Prints "The point is at (1, 2)."

# **Generic Parameters and Arguments**

This chapter describes parameters and arguments for generic types, functions, and initializers. When you declare a generic type, function, subscript, or initializer, you specify the type parameters that the generic type, function, or initializer can work with. These type parameters act as placeholders that are replaced by actual concrete type arguments when an instance of a generic type is created or a generic function or initializer is called.

For an overview of generics in Swift, see [Generics](https://docs.swift.org/swift-book/LanguageGuide/Generics.html).

## Generic Parameter Clause

A generic parameter clause specifies the type parameters of a generic type or function, along with any associated constraints and requirements on those parameters. A generic parameter clause is enclosed in angle brackets (<>) and has the following form:

1. <generic parameter list>

The generic parameter list is a comma-separated list of generic parameters, each of which has the following form:

1. type parameter: constraint

A generic parameter consists of a type parameter followed by an optional constraint. A type parameter is simply the name of a placeholder type (for example, T, U, V, Key, Value, and so on). You have access to the type parameters (and any of their associated types) in the rest of the type, function, or initializer declaration, including in the signature of the function or initializer.

The constraint specifies that a type parameter inherits from a specific class or conforms to a protocol or protocol composition. For example, in the generic function below, the generic parameter T: Comparable indicates that any type argument substituted for the type parameter T must conform to the Comparable protocol.

1. func simpleMax<T: Comparable>(\_ x: T, \_ y: T) -> T {
2. if x < y {
3. return y
4. }
5. return x
6. }

Because Int and Double, for example, both conform to the Comparable protocol, this function accepts arguments of either type. In contrast with generic types, you don’t specify a generic argument clause when you use a generic function or initializer. The type arguments are instead inferred from the type of the arguments passed to the function or initializer.

1. simpleMax(17, 42) // T is inferred to be Int
2. simpleMax(3.14159, 2.71828) // T is inferred to be Double

### **Generic Where Clauses**

You can specify additional requirements on type parameters and their associated types by including a generic where clause right before the opening curly brace of a type or function’s body. A generic where clause consists of the where keyword, followed by a comma-separated list of one or more requirements.

1. where requirements

The requirements in a generic where clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the generic where clause provides syntactic sugar for expressing simple constraints on type parameters (for example, <T: Comparable> is equivalent to <T> where T: Comparable and so on), you can use it to provide more complex constraints on type parameters and their associated types. For example, you can constrain the associated types of type parameters to conform to protocols. For example, <S: Sequence> where S.Iterator.Element: Equatable specifies that S conforms to the Sequence protocol and that the associated type S.Iterator.Element conforms to the Equatable protocol. This constraint ensures that each element of the sequence is equatable.

You can also specify the requirement that two types be identical, using the == operator. For example, <S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element expresses the constraints that S1 and S2 conform to the Sequence protocol and that the elements of both sequences must be of the same type.

Any type argument substituted for a type parameter must meet all the constraints and requirements placed on the type parameter.

You can overload a generic function or initializer by providing different constraints, requirements, or both on the type parameters. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

For more information about generic where clauses and to see an example of one in a generic function declaration, see [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192).

GRAMMAR OF A GENERIC PARAMETER CLAUSE

*generic-parameter-clause* → **<** [generic-parameter-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter-list) **>**

*generic-parameter-list* → [generic-parameter](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter) | [generic-parameter](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter) **,** [generic-parameter-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-parameter-list)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) **:** [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier)

*generic-parameter* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) **:** [protocol-composition-type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_protocol-composition-type)

*generic-where-clause* → **where** [requirement-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement-list)

*requirement-list* → [requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement) | [requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement) **,** [requirement-list](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_requirement-list)

*requirement* → [conformance-requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_conformance-requirement) | [same-type-requirement](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_same-type-requirement)

*conformance-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **:** [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier)

*conformance-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **:** [protocol-composition-type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_protocol-composition-type)

*same-type-requirement* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **==** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

## Generic Argument Clause

A generic argument clause specifies the type arguments of a generic type. A generic argument clause is enclosed in angle brackets (<>) and has the following form:

1. <generic argument list>

The generic argument list is a comma-separated list of type arguments. A type argument is the name of an actual concrete type that replaces a corresponding type parameter in the generic parameter clause of a generic type. The result is a specialized version of that generic type. The example below shows a simplified version of the Swift standard library’s generic dictionary type.

1. struct Dictionary<Key: Hashable, Value>: Collection, ExpressibleByDictionaryLiteral {
2. /\* ... \*/
3. }

The specialized version of the generic Dictionary type, Dictionary<String, Int> is formed by replacing the generic parameters Key: Hashable and Value with the concrete type arguments String and Int. Each type argument must satisfy all the constraints of the generic parameter it replaces, including any additional requirements specified in a generic where clause. In the example above, the Key type parameter is constrained to conform to the Hashable protocol and therefore String must also conform to the Hashable protocol.

You can also replace a type parameter with a type argument that is itself a specialized version of a generic type (provided it satisfies the appropriate constraints and requirements). For example, you can replace the type parameter Element in Array<Element> with a specialized version of an array, Array<Int>, to form an array whose elements are themselves arrays of integers.

1. let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

As mentioned in [Generic Parameter Clause](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#ID407), you don’t use a generic argument clause to specify the type arguments of a generic function or initializer.

# **Document Revision History**

**2019-09-10**

* Updated for Swift 5.1.
* Added information about functions that specify a protocol that their return value conforms to, instead of providing a specific named return type, to the [Opaque Types](https://docs.swift.org/swift-book/LanguageGuide/OpaqueTypes.html) chapter.
* Added information about property wrappers to the [Property Wrappers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617) section.
* Added information enumerations and structures that are frozen for library evolution to the [frozen](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID620) section.
* Added the [Functions With an Implicit Return](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID607) and [Shorthand Getter Declaration](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID608) sections with information about functions that omit return.
* Added information about using subscripts on types to the [Type Subscripts](https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html#ID609) section.
* Updated the [Enumeration Case Pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#ID424) section, now that an enumeration case pattern can match an optional value.
* Updated the [Memberwise Initializers for Structure Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID214) section, now that memberwise initializers support omitting parameters for properties that have a default value.
* Added information about dynamic members that are looked up by key path at run time to the [dynamicMemberLookup](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID585) section.
* Added macCatalyst to the list of target environments in [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539).
* Updated the [Self Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID610) section, now that Self can be used to refer to the type introduced by the current class, structure, or enumeration declaration.

**2019-03-25**

* Updated for Swift 5.0.
* Added the [Extended String Delimiters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID606) section and updated the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section with information about extended string delimiters.
* Added the [dynamicCallable](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html" \l "ID603) section with information about dynamically calling instances as functions using the dynamicCallable attribute.
* Added the [unknown](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID605) and [Switching Over Future Enumeration Cases](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID602) sections with information about handling future enumeration cases in switch statements using the unknown switch case attribute.
* Added information about the identity key path (\.self) to the [Key-Path Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID563) section.
* Added information about using the less than (<) operator in platform conditions to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.

**2018-09-17**

* Updated for Swift 4.2.
* Added information about accessing all of an enumeration’s cases to the [Iterating over Enumeration Cases](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID581) section.
* Added information about #error and #warning to the [Compile-Time Diagnostic Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID582) section.
* Added information about inlining to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section under the inlinable and usableFromInline attributes.
* Added information about members that are looked up by name at runtime to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section under the dynamicMemberLookup attribute.
* Added information about the requires\_stored\_property\_inits and warn\_unqualified\_access attributes to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* Added information about how to conditionally compile code depending on the Swift compiler version being used to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.
* Added information about #dsohandle to the [Literal Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID390) section.

**2018-03-29**

* Updated for Swift 4.1.
* Added information about synthesized implementations of equivalence operators to the [Equivalence Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID45) section.
* Added information about conditional protocol conformance to the [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter, and to the [Conditionally Conforming to a Protocol](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID574) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter.
* Added information about recursive protocol constraints to the [Using a Protocol in Its Associated Type’s Constraints](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID575) section.
* Added information about the canImport() and targetEnvironment() platform conditions to [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539).

**2017-12-04**

* Updated for Swift 4.0.3.
* Updated the [Key-Path Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID563) section, now that key paths support subscript components.

**2017-09-19**

* Updated for Swift 4.0.
* Added information about exclusive access to memory to the [Memory Safety](https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html) chapter.
* Added the [Associated Types with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID557) section, now that you can use generic where clauses to constrain associated types.
* Added information about multiline string literals to the [String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID286) section of the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter, and to the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section of the [Lexical Structure](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html) chapter.
* Updated the discussion of the objc attribute in [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348), now that this attribute is inferred in fewer places.
* Added the [Generic Subscripts](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID558) section, now that subscripts can be generic.
* Updated the discussion in the [Protocol Composition](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID282) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter, and in the [Protocol Composition Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID454) section of the [Types](https://docs.swift.org/swift-book/ReferenceManual/Types.html) chapter, now that protocol composition types can contain a superclass requirement.
* Updated the discussion of protocol extensions in [Extension Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID378) now that final isn’t allowed in them.
* Added information about preconditions and fatal errors to the [Assertions and Preconditions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335) section.

**2017-03-27**

* Updated for Swift 3.1.
* Added the [Extensions with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID553) section with information about extensions that include requirements.
* Added examples of iterating over a range to the [For-In Loops](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID121) section.
* Added an example of failable numeric conversions to the [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224) section.
* Added information to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section about using the available attribute with a Swift language version.
* Updated the discussion in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section to note that argument labels are not allowed when writing a function type.
* Updated the discussion of Swift language version numbers in the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section, now that an optional patch number is allowed.
* Updated the discussion in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section, now that Swift distinguishes between functions that take multiple parameters and functions that take a single parameter of a tuple type.
* Removed the Dynamic Type Expression section from the [Expressions](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html) chapter, now that type(of:) is a Swift standard library function.

**2016-10-27**

* Updated for Swift 3.0.1.
* Updated the discussion of weak and unowned references in the [Automatic Reference Counting](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html) chapter.
* Added information about the unowned, unowned(safe), and unowned(unsafe) declaration modifiers in the [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381) section.
* Added a note to the [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342) section about using an optional value when a value of type Any is expected.
* Updated the [Expressions](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html) chapter to separate the discussion of parenthesized expressions and tuple expressions.

**2016-09-13**

* Updated for Swift 3.0.
* Updated the discussion of functions in the [Functions](https://docs.swift.org/swift-book/LanguageGuide/Functions.html) chapter and the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section to note that all parameters get an argument label by default.
* Updated the discussion of operators in the [Advanced Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html) chapter, now that you implement them as type methods instead of as global functions.
* Added information about the open and fileprivate access-level modifiers to the [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html) chapter.
* Updated the discussion of inout in the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section to note that it appears in front of a parameter’s type instead of in front of a parameter’s name.
* Updated the discussion of the @noescape and @autoclosure attributes in the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) and [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) sections and the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter now that they are type attributes, rather than declaration attributes.
* Added information about operator precedence groups to the [Precedence for Custom Infix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID47) section of the [Advanced Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html) chapter, and to the [Precedence Group Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID550) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Updated discussion throughout to use macOS instead of OS X, Error instead of ErrorProtocol, and protocol names such as ExpressibleByStringLiteral instead of StringLiteralConvertible.
* Updated the discussion in the [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192) section of the [Generics](https://docs.swift.org/swift-book/LanguageGuide/Generics.html) chapter and in the [Generic Parameters and Arguments](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html) chapter, now that generic where clauses are written at the end of a declaration.
* Updated the discussion in the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) section, now that closures are nonescaping by default.
* Updated the discussion in the [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333) section of the [The Basics](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html) chapter and the [While Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID432) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter, now that if, while, and guard statements use a comma-separated list of conditions without where clauses.
* Added information about switch cases that have multiple patterns to the [Switch](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID129) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Switch Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID436) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Updated the discussion of function types in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section now that function argument labels are no longer part of a function’s type.
* Updated the discussion of protocol composition types in the [Protocol Composition](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID282) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter and in the [Protocol Composition Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID454) section of the [Types](https://docs.swift.org/swift-book/ReferenceManual/Types.html) chapter to use the new Protocol1 & Protocol2 syntax.
* Updated the discussion in the Dynamic Type Expression section to use the new type(of:) syntax for dynamic type expressions.
* Updated the discussion of line control statements to use the #sourceLocation(file:line:) syntax in the [Line Control Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID540) section.
* Updated the discussion in [Functions that Never Return](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID551) to use the new Never type.
* Added information about playground literals to the [Literal Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID390) section.
* Updated the discussion in the [In-Out Parameters](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545) section to note that only nonescaping closures can capture in-out parameters.
* Updated the discussion about default parameters in the [Default Parameter Values](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID169) section, now that they can’t be reordered in function calls.
* Updated attribute arguments to use a colon in the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter.
* Added information about throwing an error inside the catch block of a rethrowing function to the [Rethrowing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID531) section.
* Added information about accessing the selector of an Objective-C property’s getter or setter to the [Selector Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID547) section.
* Added information to the [Type Alias Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID361) section about generic type aliases and using type aliases inside of protocols.
* Updated the discussion of function types in the [Function Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#ID449) section to note that parentheses around the parameter types are required.
* Updated the [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html) chapter to note that the @IBAction, @IBOutlet, and @NSManaged attributes imply the @objc attribute.
* Added information about the @GKInspectable attribute to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* Updated the discussion of optional protocol requirements in the [Optional Protocol Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID284) section to clarify that they are used only in code that interoperates with Objective-C.
* Removed the discussion of explicitly using let in function parameters from the [Function Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID362) section.
* Removed the discussion of the Boolean protocol from the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter, now that the protocol has been removed from the Swift standard library.
* Corrected the discussion of the @NSApplicationMain attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.

**2016-03-21**

* Updated for Swift 2.2.
* Added information about how to conditionally compile code depending on the version of Swift being used to the [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) section.
* Added information about how to distinguish between methods or initializers whose names differ only by the names of their arguments to the [Explicit Member Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID400) section.
* Added information about the #selector syntax for Objective-C selectors to the [Selector Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID547) section.
* Updated the discussion of associated types to use the associatedtype keyword in the [Associated Types](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID189) and [Protocol Associated Type Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID374) sections.
* Updated information about initializers that return nil before the instance is fully initialized in the [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224) section.
* Added information about comparing tuples to the [Comparison Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID70) section.
* Added information about using keywords as external parameter names to the [Keywords and Punctuation](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID413) section.
* Updated the discussion of the @objc attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section to note that enumerations and enumeration cases can use this attribute.
* Updated the [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) section with discussion of custom operators that contain a dot.
* Added a note to the [Rethrowing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID531) section that rethrowing functions can’t directly throw errors.
* Added a note to the [Property Observers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262) section about property observers being called when you pass a property as an in-out parameter.
* Added a section about error handling to the [A Swift Tour](https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html) chapter.
* Updated figures in the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section to show the deallocation process more clearly.
* Removed discussion of C-style for loops, the ++ prefix and postfix operators, and the -- prefix and postfix operators.
* Removed discussion of variable function arguments and the special syntax for curried functions.

**2015-10-20**

* Updated for Swift 2.1.
* Updated the [String Interpolation](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID292) and [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) sections now that string interpolations can contain string literals.
* Added the [Escaping Closures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID546) section with information about the @noescape attribute.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) and [Conditional Compilation Block](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID539) sections with information about tvOS.
* Added information about the behavior of in-out parameters to the [In-Out Parameters](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545) section.
* Added information to the [Capture Lists](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID544) section about how values specified in closure capture lists are captured.
* Updated the [Accessing Properties Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID248) section to clarify how assignment through optional chaining behaves.
* Improved the discussion of autoclosures in the [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) section.
* Added an example that uses the ?? operator to the [A Swift Tour](https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html) chapter.

**2015-09-16**

* Updated for Swift 2.0.
* Added information about error handling to the [Error Handling](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html) chapter, the [Do Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID533) section, the [Throw Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID518) section, the [Defer Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID532) section, and the [Try Operator](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID516) section.
* Updated the [Representing and Throwing Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID509) section, now that all types can conform to the ErrorType protocol.
* Added information about the new try? keyword to the [Converting Errors to Optional Values](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID542) section.
* Added information about recursive enumerations to the [Recursive Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID536) section of the [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html) chapter and the [Enumerations with Cases of Any Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID365) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Added information about API availability checking to the [Checking API Availability](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID523) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Availability Condition](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID522) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Added information about the new guard statement to the [Early Exit](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID525) section of the [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html) chapter and the [Guard Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID524) section of the [Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html) chapter.
* Added information about protocol extensions to the [Protocol Extensions](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID521) section of the [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html) chapter.
* Added information about access control for unit testing to the [Access Levels for Unit Test Targets](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID519) section of the [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html) chapter.
* Added information about the new optional pattern to the [Optional Pattern](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html#ID520) section of the [Patterns](https://docs.swift.org/swift-book/ReferenceManual/Patterns.html) chapter.
* Updated the [Repeat-While](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID126) section with information about the repeat-while loop.
* Updated the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter, now that String no longer conforms to the CollectionType protocol from the Swift standard library.
* Added information about the new Swift standard library print(\_:separator:terminator) function to the [Printing Constants and Variables](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID314) section.
* Added information about the behavior of enumeration cases with String raw values to the [Implicitly Assigned Raw Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID535) section of the [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html) chapter and the [Enumerations with Cases of a Raw-Value Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID366) section of the [Declarations](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html) chapter.
* Added information about the @autoclosure attribute—including its @autoclosure(escaping) form—to the [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html" \l "ID543) section.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section with information about the @available and @warn\_unused\_result attributes.
* Updated the [Type Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID350) section with information about the @convention attribute.
* Added an example of using multiple optional bindings with a where clause to the [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333) section.
* Added information to the [String Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID417) section about how concatenating string literals using the + operator happens at compile time.
* Added information to the [Metatype Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html" \l "ID455) section about comparing metatype values and using them to construct instances with initializer expressions.
* Added a note to the [Debugging with Assertions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID336) section about when user-defined assertions are disabled.
* Updated the discussion of the @NSManaged attribute in the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section, now that the attribute can be applied to certain instance methods.
* Updated the [Variadic Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID171) section, now that variadic parameters can be declared in any position in a function’s parameter list.
* Added information to the [Overriding a Failable Initializer](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID229) section about how a nonfailable initializer can delegate up to a failable initializer by force-unwrapping the result of the superclass’s initializer.
* Added information about using enumeration cases as functions to the [Enumerations with Cases of Any Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID365) section.
* Added information about explicitly referencing an initializer to the [Initializer Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID399) section.
* Added information about build configuration and line control statements to the [Compiler Control Statements](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID538) section.
* Added a note to the [Metatype Type](https://docs.swift.org/swift-book/ReferenceManual/Types.html" \l "ID455) section about constructing class instances from metatype values.
* Added a note to the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section about weak references being unsuitable for caching.
* Updated a note in the [Type Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264) section to mention that stored type properties are lazily initialized.
* Updated the [Capturing Values](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID103) section to clarify how variables and constants are captured in closures.
* Updated the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section to describe when you can apply the @objc attribute to classes.
* Added a note to the [Handling Errors](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#ID512) section about the performance of executing a throw statement. Added similar information about the do statement in the [Do Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID533) section.
* Updated the [Type Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264) section with information about stored and computed type properties for classes, structures, and enumerations.
* Updated the [Break Statement](https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID441) section with information about labeled break statements.
* Updated a note in the [Property Observers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262) section to clarify the behavior of willSet and didSet observers.
* Added a note to the [Access Levels](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID5) section with information about the scope of private access.
* Added a note to the [Weak References](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID53) section about the differences in weak references between garbage collected systems and ARC.
* Updated the [Special Characters in String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID295) section with a more precise definition of Unicode scalars.

**2015-04-08**

* Updated for Swift 1.2.
* Swift now has a native Set collection type. For more information, see [Sets](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID484).
* @autoclosure is now an attribute of the parameter declaration, not its type. There is also a new @noescape parameter declaration attribute. For more information, see [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348).
* Type methods and properties now use the static keyword as a declaration modifier. For more information see [Type Variable Properties](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID483).
* Swift now includes the as? and as! failable downcast operators. For more information, see [Checking for Protocol Conformance](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID283).
* Added a new guide section about [String Indices](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID534).
* Removed the overflow division (&/) and overflow remainder (&%) operators from [Overflow Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID37).
* Updated the rules for constant and constant property declaration and initialization. For more information, see [Constant Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID355).
* Updated the definition of Unicode scalars in string literals. See [Special Characters in String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID295).
* Updated [Range Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID73) to note that a half-open range with the same start and end index will be empty.
* Updated [Closures Are Reference Types](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID104) to clarify the capturing rules for variables.
* Updated [Value Overflow](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID38) to clarify the overflow behavior of signed and unsigned integers
* Updated [Protocol Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID369) to clarify protocol declaration scope and members.
* Updated [Defining a Capture List](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID58) to clarify the syntax for weak and unowned references in closure capture lists.
* Updated [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) to explicitly mention examples of supported characters for custom operators, such as those in the Mathematical Operators, Miscellaneous Symbols, and Dingbats Unicode blocks.
* Constants can now be declared without being initialized in local function scope. They must have a set value before first use. For more information, see [Constant Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID355).
* In an initializer, constant properties can now only assign a value once. For more information, see [Assigning Constant Properties During Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID212).
* Multiple optional bindings can now appear in a single if statement as a comma-separated list of assignment expressions. For more information, see [Optional Binding](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID333).
* An [Optional-Chaining Expression](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID405) must appear within a postfix expression.
* Protocol casts are no longer limited to @objc protocols.
* Type casts that can fail at runtime now use the as? or as! operator, and type casts that are guaranteed not to fail use the as operator. For more information, see [Type-Casting Operators](https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#ID388).

**2014-10-16**

* Updated for Swift 1.1.
* Added a full guide to [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID224).
* Added a description of [Failable Initializer Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html" \l "ID274) for protocols.
* Constants and variables of type Any can now contain function instances. Updated the example in [Type Casting for Any and AnyObject](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID342) to show how to check for and cast to a function type within a switch statement.
* Enumerations with raw values now have a rawValue property rather than a toRaw() method and a failable initializer with a rawValue parameter rather than a fromRaw() method. For more information, see [Raw Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID149) and [Enumerations with Cases of a Raw-Value Type](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID366).
* Added a new reference section about [Failable Initializers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html" \l "ID376), which can trigger initialization failure.
* Custom operators can now contain the ? character. Updated the [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418) reference to describe the revised rules. Removed a duplicate description of the valid set of operator characters from [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46).

**2014-08-18**

* New document that describes Swift 1.0, Apple’s new programming language for building iOS and OS X apps.
* Added a new section about [Initializer Requirements](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID272) in protocols.
* Added a new section about [Class-Only Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID281).
* [Assertions and Preconditions](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335) can now use string interpolation. Removed a note to the contrary.
* Updated the [Concatenating Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID291) section to reflect the fact that String and Character values can no longer be combined with the addition operator (+) or addition assignment operator (+=). These operators are now used only with String values. Use the String type’s append(\_:) method to append a single Character value onto the end of a string.
* Added information about the availability attribute to the [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348) section.
* [Optionals](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID330) no longer implicitly evaluate to true when they have a value and false when they do not, to avoid confusion when working with optional Bool values. Instead, make an explicit check against nil with the == or != operators to find out if an optional contains a value.
* Swift now has a [Nil-Coalescing Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID72) (a ?? b), which unwraps an optional’s value if it exists, or returns a default value if the optional is nil.
* Updated and expanded the [Comparing Strings](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID298) section to reflect and demonstrate that string and character comparison and prefix / suffix comparison are now based on Unicode canonical equivalence of extended grapheme clusters.
* You can now try to set a property’s value, assign to a subscript, or call a mutating method or operator through [Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html). The information about [Accessing Properties Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID248) has been updated accordingly, and the examples of checking for method call success in [Calling Methods Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID249) have been expanded to show how to check for property setting success.
* Added a new section about [Accessing Subscripts of Optional Type](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID251) through optional chaining.
* Updated the [Accessing and Modifying an Array](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID110) section to note that you can no longer append a single item to an array with the += operator. Instead, use the append(\_:) method, or append a single-item array with the += operator.
* Added a note that the start value a for the [Range Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID73) a...b and a..<b must not be greater than the end value b.
* Rewrote the [Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html) chapter to remove its introductory coverage of initializer overrides. This chapter now focuses more on the addition of new functionality in a subclass, and the modification of existing functionality with overrides. The chapter’s example of [Overriding Property Getters and Setters](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html#ID200) has been rewritten to show how to override a description property. (The examples of modifying an inherited property’s default value in a subclass initializer have been moved to the [Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html) chapter.)
* Updated the [Initializer Inheritance and Overriding](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID221) section to note that overrides of a designated initializer must now be marked with the override modifier.
* Updated the [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231) section to note that the required modifier is now written before every subclass implementation of a required initializer, and that the requirements for required initializers can now be satisfied by automatically inherited initializers.
* Infix [Operator Methods](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID42) no longer require the @infix attribute.
* The @prefix and @postfix attributes for [Prefix and Postfix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID43) have been replaced by prefix and postfix declaration modifiers.
* Added a note about the order in which [Prefix and Postfix Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID43) are applied when both a prefix and a postfix operator are applied to the same operand.
* Operator functions for [Compound Assignment Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID44) no longer use the @assignment attribute when defining the function.
* The order in which modifiers are specified when defining [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46) has changed. You now write prefix operator rather than operator prefix, for example.
* Added information about the dynamic declaration modifier in [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381).
* Added information about how type inference works with [Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID414).
* Added more information about curried functions.
* Added a new chapter about [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html).
* Updated the [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) chapter to reflect the fact that Swift’s Character type now represents a single Unicode extended grapheme cluster. Includes a new section on [Extended Grapheme Clusters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID296) and more information about [Unicode Scalar Values](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID294) and [Comparing Strings](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID298).
* Updated the [String Literals](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID286) section to note that Unicode scalars inside string literals are now written as \u{n}, where n is a hexadecimal number between 0 and 10FFFF, the range of Unicode’s codespace.
* The NSString length property is now mapped onto Swift’s native String type as utf16Count, not utf16count.
* Swift’s native String type no longer has an uppercaseString or lowercaseString property. The corresponding section in [Strings and Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html) has been removed, and various code examples have been updated.
* Added a new section about [Initializer Parameters Without Argument Labels](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID210).
* Added a new section about [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231).
* Added a new section about [Optional Tuple Return Types](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID165).
* Updated the [Type Annotations](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID312) section to note that multiple related variables can be defined on a single line with one type annotation.
* The @optional, @lazy, @final, and @required attributes are now the optional, lazy, final, and required [Declaration Modifiers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID381).
* Updated the entire book to refer to ..< as the [Half-Open Range Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID75) (rather than the “half-closed range operator”).
* Updated the [Accessing and Modifying a Dictionary](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID116) section to note that Dictionary now has a Boolean isEmpty property.
* Clarified the full list of characters that can be used when defining [Custom Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID46).
* nil and the Booleans true and false are now [Literals](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID414).
* Swift’s Array type now has full value semantics. Updated the information about [Mutability of Collections](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID106) and [Arrays](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID107) to reflect the new approach. Also clarified the assignment and copy behavior for strings arrays and dictionaries.
* [Array Type Shorthand Syntax](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID108) is now written as [SomeType] rather than SomeType[].
* Added a new section about [Dictionary Type Shorthand Syntax](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID114), which is written as [KeyType: ValueType].
* Added a new section about [Hash Values for Set Types](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID493).
* Examples of [Closure Expressions](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID95) now use the global sorted(\_:\_:) function rather than the global sort(\_:\_:) function, to reflect the new array value semantics.
* Updated the information about [Memberwise Initializers for Structure Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html" \l "ID214) to clarify that the memberwise structure initializer is made available even if a structure’s stored properties do not have default values.
* Updated to ..< rather than .. for the [Half-Open Range Operator](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID75).
* Added an example of [Extending a Generic Type](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID185).

Version Control System

**Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS −

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

Distributed Version Control System

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

## What is a Git repository?

A [Git repository](http://bitbucket.org/code-repository) is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

**Note:** You can create public repositories for an open source project. When creating your public repository, make sure to include a [license file](http://choosealicense.com/) that determines how you want your project to be shared with others. For more information on open source, specifically how to create and grow an open source project, we've created [Open Source Guides](https://opensource.guide/) that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your open source project. You can also take a free [GitHub Learning Lab](https://lab.github.com/) course on maintaining open source communities.

# **Adding an existing project to GitHub using the command line**

Putting your existing work on GitHub can let you share and collaborate in lots of great ways.

[Create a new repository](https://help.github.com/en/articles/creating-a-new-repository) on GitHub. To avoid errors, do not initialize the new repository with README, license, or gitignore files. You can add these files after your project has been pushed to GitHub.

1. Open Terminal.
2. Change the current working directory to your local project.
3. Initialize the local directory as a Git repository.

$ git init

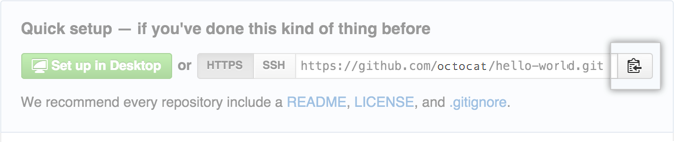
1. Add the files in your new local repository. This stages them for the first commit.
2. $ git add .

# Adds the files in the local repository and stages them for commit. To unstage a file, use 'git reset HEAD YOUR-FILE'.

1. Commit the files that you've staged in your local repository.
2. $ git commit -m "First commit"

# Commits the tracked changes and prepares them to be pushed to a remote repository. To remove this commit and modify the file, use 'git reset --soft HEAD~1' and commit and add the file again.

1. At the top of your GitHub repository's Quick Setup page, click  to copy the remote repository URL.



1. In Terminal, [add the URL for the remote repository](https://help.github.com/en/articles/adding-a-remote) where your local repository will be pushed.
2. $ git remote add origin remote repository URL# Sets the new remote
3. $ git remote -v

# Verifies the new remote URL

1. [Push the changes](https://help.github.com/en/articles/pushing-commits-to-a-remote-repository) in your local repository to GitHub.
2. $ git push -u origin master

# Pushes the changes in your local repository up to the remote

<https://help.github.com/en/github/importing-your-projects-to-github/adding-an-existing-project-to-github-using-the-command-line>

<https://help.github.com/en/github/getting-started-with-github/create-a-repo>

<https://help.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository>

<https://help.github.com/en/github/administering-a-repository/deleting-a-repository>

<https://stackoverflow.com/questions/8461528/replace-github-repo-while-preserving-issues-wiki-etc>

<https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-and-deleting-branches-within-your-repository>

<https://stackoverflow.com/questions/1911109/how-do-i-clone-a-specific-git-branch>

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The git pull command is actually a combination of two other commands, [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) followed by [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a master branch and a remote origin.

## Common Options

git pull <remote>

Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as git fetch <remote> followed by git merge origin/<current-branch>.

git pull --no-commit <remote>

Similar to the default invocation, fetches the remote content but does not create a new merge commit.

git pull --rebase <remote>

Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

git pull --verbose

Gives verbose output during a pull which displays the content being downloaded and the merge details.