ON THIS PAGE

Swift

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.

> GRAMMAR OF A STATEMENT
>
> *statement* → expression **;** *opt*
> *statement* → declaration **;** *opt*
> *statement* → loop-statement **;** *opt*
> *statement* → branch-statement **;** *opt*
> *statement* → labeled-statement **;** *opt*
> *statement* → control-transfer-statement **;** *opt*
> *statement* → defer-statement **;** *opt*

> *statement* → do-statement **;** *opt*
> *statement* → compiler-control-statement
> *statements* → statement statements *opt*

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 and Continue Statement below.

> GRAMMAR OF A LOOP STATEMENT
>
> *loop-statement* → for-in-statement
> *loop-statement* → while-statement
> *loop-statement* → 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` protocol.

A `for-in` statement has the following form:

```
for item in collection {
    statements
}
```

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` 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 **`in`** expression where-clause$_{opt}$ 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:

```
while condition {
    statements
}
```

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.

2. 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.

> GRAMMAR OF A WHILE STATEMENT

> *while-statement* → **`while`** condition-list code-block
>
> *condition-list* → condition | condition **`,`** condition-list
> *condition* → expression | availability-condition | case-condition | optional-binding-condition
>
> *case-condition* → **`case`** pattern initializer
> *optional-binding-condition* → **`let`** pattern initializer | **`var`** pattern 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:

```
repeat {
    statements
} 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.

> GRAMMAR OF A REPEAT-WHILE STATEMENT
>
> *repeat-while-statement* → **`repeat`** code-block **`while`** 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 below.

> GRAMMAR OF A BRANCH STATEMENT
>
> *branch-statement* → if-statement
> *branch-statement* → guard-statement
> *branch-statement* → 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:

```
if condition {
    statements
}
```

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:

```
if condition {
```

```
    statements to execute if condition is true
} else {
    statements to execute if condition is false
}
```

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:

```
if condition 1 {
    statements to execute if condition 1 is true
} else if condition 2 {
    statements to execute if condition 2 is true
} else {
    statements to execute if both conditions are false
}
```

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.

> GRAMMAR OF AN IF STATEMENT
>
> *if-statement* → **if** condition-list code-block else-clause*opt*
>
> *else-clause* → **else** code-block | **else** 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:

```
guard condition else {
    statements
}
```

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.

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 below. For more information on functions with the `Never` return type, see Functions that Never Return.

> GRAMMAR OF A GUARD STATEMENT
>
> *guard-statement* → **`guard`** condition-list **`else`** 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:

```
switch control expression {
case pattern 1:
    statements
case pattern 2 where condition:
    statements
case pattern 3 where condition,
     pattern 4 where condition:
```

```
        statements
    default:
        statements
}
```

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 in Control Flow.

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)`.

```
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. Only the standard library, Swift overlays for Apple frameworks, and C and Objective-C code can declare nonfrozen enumerations. Enumerations you declare in Swift can't be nonfrozen.

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`] 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.

```
let representation: Mirror.AncestorRepresentation = .generated
switch representation {
case .customized:
    print("Use the nearest ancestor’s implementation.")
case .generated:
    print("Generate a default mirror for all ancestor classes.")
case .suppressed:
    print("Suppress the representation of all ancestor classes.")
@unknown default:
    print("Use a representation that was unknown when this code was compiled.")
}
// 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 below.

GRAMMAR OF A SWITCH STATEMENT

switch-statement → **switch** expression **{** switch-cases*opt* **}**
switch-cases → switch-case switch-cases*opt*
switch-case → case-label statements
switch-case → default-label statements
switch-case → conditional-switch-case

case-label → attributes*opt* **case** case-item-list **:**
case-item-list → pattern where-clause*opt* | pattern where-clause*opt* **,** case-item-list
default-label → attributes*opt* **default** **:**

where-clause → **where** where-expression
where-expression → expression

conditional-switch-case → switch-if-directive-clause switch-elseif-directive-clauses*opt* switch-else-directive-clause*opt* endif-directive
switch-if-directive-clause → if-directive compilation-condition switch-cases*opt*
switch-elseif-directive-clauses → elseif-directive-clause switch-elseif-directive-clauses*opt*
switch-elseif-directive-clause → elseif-directive compilation-condition switch-cases*opt*
switch-else-directive-clause → else-directive 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 and Continue Statement 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 in Control Flow.

> GRAMMAR OF A LABELED STATEMENT
>
> *labeled-statement* → statement-label loop-statement
> *labeled-statement* → statement-label if-statement
> *labeled-statement* → statement-label switch-statement
> *labeled-statement* → statement-label do-statement
>
> *statement-label* → label-name **:**
> *label-name* → 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
> *control-transfer-statement* → continue-statement
> *control-transfer-statement* → fallthrough-statement
> *control-transfer-statement* → return-statement
> *control-transfer-statement* → 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.

```
break
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 and Labeled Statements in Control Flow.

> GRAMMAR OF A BREAK STATEMENT
>
> *break-statement* → **`break`** 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.

```
continue
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 and Labeled Statements in Control Flow.

> GRAMMAR OF A CONTINUE STATEMENT
>
> *continue-statement* → **`continue`** 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 in Control Flow.

> 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.

```
return
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, 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*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.

```
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 in Error Handling.

GRAMMAR OF A THROW STATEMENT

throw-statement → **`throw`** 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:

```
defer {
    statements
}
```

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.

```
func f() {
    defer { print("First defer") }
    defer { print("Second defer") }
    print("End of function")
}
f()
// Prints "End of function"
// Prints "Second defer"
// 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

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:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

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.

To see an example of how to use a `do` statement with several `catch` clauses, see Handling Errors.

> GRAMMAR OF A DO STATEMENT
>
> *do-statement* → **do** code-block catch-clauses$_{opt}$
> *catch-clauses* → catch-clause catch-clauses$_{opt}$
> *catch-clause* → **catch** pattern$_{opt}$ where-clause$_{opt}$ 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
> *compiler-control-statement* → line-control-statement
> *compiler-control-statement* → 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:

```
#if compilation condition
statements
```

```
#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`

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:

```
#if compiler(>=5)
print("Compiled with the Swift 5 compiler or later")
#endif
#if swift(>=4.2)
```

```
print("Compiled in Swift 4.2 mode or later")
#endif
#if compiler(>=5) && swift(<5)
print("Compiled with the Swift 5 compiler or later in a
  Swift mode earlier than 5")
#endif
// Prints "Compiled with the Swift 5 compiler or later"
// Prints "Compiled in Swift 4.2 mode or later"
// 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:

```
#if compilation condition 1
statements to compile if compilation condition 1 is true
#elseif compilation condition 2
statements to compile if compilation condition 2 is true
```

```
#else
    statements to compile if both compilation conditions are false

#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 elseif-directive-clauses$_{opt}$ else-directive-clause$_{opt}$ endif-directive

if-directive-clause → if-directive compilation-condition statements$_{opt}$
elseif-directive-clauses → elseif-directive-clause elseif-directive-clauses$_{opt}$
elseif-directive-clause → elseif-directive compilation-condition statements$_{opt}$
else-directive-clause → else-directive statements$_{opt}$
if-directive → **#if**
elseif-directive → **#elseif**
else-directive → **#else**
endif-directive → **#endif**

compilation-condition → platform-condition
compilation-condition → identifier
compilation-condition → boolean-literal
compilation-condition → **(** compilation-condition **)**
compilation-condition → **!** compilation-condition
compilation-condition → compilation-condition **&&** compilation-condition
compilation-condition → compilation-condition **||** compilation-condition

platform-condition → **os** **(** operating-system **)**
platform-condition → **arch** **(** architecture **)**
platform-condition → **swift** **(** **>=** swift-version **)** | **swift** **(** **<** swift-version **)**
platform-condition → **compiler** **(** **>=** swift-version **)** | **compiler** **(** **<** swift-version **)**
platform-condition → **canImport** **(** module-name **)**

platform-condition → **targetEnvironment (** environment **)**

operating-system → **macOS** | **iOS** | **watchOS** | **tvOS**
architecture → **i386** | **x86_64** | **arm** | **arm64**
swift-version → decimal-digits swift-version-continuation*opt*
swift-version-continuation → **.** decimal-digits swift-version-continuation*opt*
module-name → 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:

```
#sourceLocation(file: filename, line: line number)
#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 **, line:** line-number **)**
line-control-statement → **#sourceLocation ( )**
line-number → A decimal integer greater than zero
file-name → 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:

```
#error("error message")
#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 **)**
> *diagnostic-statement* → **#warning** **(** diagnostic-message **)**
>
> *diagnostic-message* → 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:

```
if #available(platform name version, ..., *) {
    statements to execute if the APIs are available
} else {

    fallback statements to execute if the APIs are unavailable

}
```

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.

Unlike Boolean conditions, you can't combine availability conditions using logical operators such as `&&` and `||`.

GRAMMAR OF AN AVAILABILITY CONDITION

availability-condition → **`#available`** **`(`** availability-arguments **`)`**
availability-arguments → availability-argument | availability-argument **`,`** availability-arguments
availability-argument → platform-name platform-version
availability-argument → **`*`**

platform-name → **`iOS`** | **`iOSApplicationExtension`**
platform-name → **`macOS`** | **`macOSApplicationExtension`**
platform-name → **`watchOS`**
platform-name → **`tvOS`**
platform-version → decimal-digits
platform-version → decimal-digits **`.`** decimal-digits
platform-version → decimal-digits **`.`** decimal-digits **`.`** decimal-digits

< Expressions Declarations >