ON THIS PAGE

Swift

Declarations

A *declaration* introduces a new name or construct into your program. For example, you use declarations to introduce functions and methods, to introduce variables and constants, and to define enumeration, structure, class, and protocol types. You can also use a declaration to extend the behavior of an existing named type and to import symbols into your program that are declared elsewhere.

In Swift, most declarations are also definitions in the sense that they are implemented or initialized at the same time they are declared. That said, because protocols don't implement their members, most protocol members are declarations only. For convenience and because the distinction isn't that important in Swift, the term *declaration* covers both declarations and definitions.

GRAMMAR OF A DECLARATION

declaration → import-declaration
declaration → constant-declaration
declaration → variable-declaration
declaration → typealias-declaration
declaration → function-declaration
declaration → enum-declaration
declaration → struct-declaration
declaration → class-declaration
declaration → protocol-declaration
declaration → initializer-declaration
declaration → deinitializer-declaration
declaration → extension-declaration
declaration → subscript-declaration
declaration → operator-declaration
declaration → precedence-group-declaration

> *declarations* → declaration declarations$_{opt}$

Top-Level Code

The top-level code in a Swift source file consists of zero or more statements, declarations, and expressions. By default, variables, constants, and other named declarations that are declared at the top-level of a source file are accessible to code in every source file that is part of the same module. You can override this default behavior by marking the declaration with an access-level modifier, as described in Access Control Levels.

> GRAMMAR OF A TOP-LEVEL DECLARATION
>
> *top-level-declaration* → statements$_{opt}$

Code Blocks

A *code block* is used by a variety of declarations and control structures to group statements together. It has the following form:

```
{
    statements
}
```

The *statements* inside a code block include declarations, expressions, and other kinds of statements and are executed in order of their appearance in source code.

> GRAMMAR OF A CODE BLOCK
>
> *code-block* → **{** statements$_{opt}$ **}**

Import Declaration

An *import declaration* lets you access symbols that are declared outside the current file. The basic form imports the entire module; it consists of the `import` keyword followed by a module name:

```
import module
```

Providing more detail limits which symbols are imported—you can specify a specific submodule or a specific declaration within a module or submodule. When this detailed form is used, only the imported symbol (and not the module that declares it) is made available in the current scope.

```
import import kind module.symbol name
import module.submodule
```

GRAMMAR OF AN IMPORT DECLARATION

import-declaration → attributes$_{opt}$ **import** import-kind$_{opt}$ import-path

import-kind → **typealias** | **struct** | **class** | **enum** | **protocol** | **let** | **var** | **func**

import-path → import-path-identifier | import-path-identifier **.** import-path

import-path-identifier → identifier | operator

Constant Declaration

A *constant declaration* introduces a constant named value into your program. Constant declarations are declared using the `let` keyword and have the following form:

```
let constant name: type = expression
```

A constant declaration defines an immutable binding between the *constant name* and the value of the initializer *expression*; after the value of a constant is set, it cannot be changed. That said, if a constant is initialized with a class object, the object itself can change, but the binding between the constant name and the object it refers to can't.

When a constant is declared at global scope, it must be initialized with a value. When a constant declaration occurs in the context of a function or method, it can be initialized later, as long as it is guaranteed to have a value set before the first time its value is read. When a constant declaration occurs in the context of a class or structure declaration, it is considered a *constant property*. Constant declarations are not computed properties and therefore do not have getters or setters.

If the *constant name* of a constant declaration is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

```
let (firstNumber, secondNumber) = (10, 42)
```

In this example, `firstNumber` is a named constant for the value `10`, and `secondNumber` is a named constant for the value `42`. Both constants can now be used independently:

```
print("The first number is \(firstNumber).")
// Prints "The first number is 10."
print("The second number is \(secondNumber).")
// Prints "The second number is 42."
```

The type annotation (`:` *type*) is optional in a constant declaration when the type of the *constant name* can be inferred, as described in Type Inference.

To declare a constant type property, mark the declaration with the `static` declaration modifier. A constant type property of a class is always implicitly final; you can't mark it with the `class` or `final` declaration modifier to allow or disallow overriding by subclasses. Type properties are discussed in Type Properties.

For more information about constants and for guidance about when to use them, see Constants and Variables and Stored Properties.

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → attributes$_{opt}$ declaration-modifiers$_{opt}$ **let** pattern-initializer-list

pattern-initializer-list → pattern-initializer | pattern-initializer **,** pattern-initializer-list

pattern-initializer → pattern initializer$_{opt}$

initializer → **=** expression

Variable Declaration

A *variable declaration* introduces a variable named value into your program and is declared using the `var` keyword.

Variable declarations have several forms that declare different kinds of named, mutable values, including stored and computed variables and properties, stored variable and property observers, and static variable properties. The appropriate form to use depends on the scope at which the variable is declared and the kind of variable you intend to declare.

> NOTE
>
> You can also declare properties in the context of a protocol declaration, as described in Protocol Property Declaration.

You can override a property in a subclass by marking the subclass's property declaration with the `override` declaration modifier, as described in Overriding.

Stored Variables and Stored Variable Properties

The following form declares a stored variable or stored variable property:

```
var variable name: type = expression
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it is referred to as a *stored variable*. When it is declared in the context of a class or structure declaration, it is referred to as a *stored variable property*.

The initializer *expression* can't be present in a protocol declaration, but in all other contexts, the initializer *expression* is optional. That said, if no initializer *expression* is present, the variable declaration must include an explicit type annotation (`:` *type*).

As with constant declarations, if the *variable name* is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

As their names suggest, the value of a stored variable or a stored variable property is stored in memory.

Computed Variables and Computed Properties

The following form declares a computed variable or computed property:

```
var variable name: type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class, structure, enumeration, or extension declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it is referred to as a *computed variable*. When it is declared in the context of a class, structure, or extension declaration, it is referred to as a *computed property*.

The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly, as described in Read-Only Computed Properties. But if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses is optional. If you provide a setter name, it is used as the name of the parameter to the setter. If you do not provide a setter name, the default parameter name to the setter is `newValue`, as described in Shorthand Setter Declaration.

Unlike stored named values and stored variable properties, the value of a computed named value or a computed property is not stored in memory.

For more information and to see examples of computed properties, see Computed Properties.

Stored Variable Observers and Property Observers

You can also declare a stored variable or property with `willSet` and `didSet` observers. A stored variable or property declared with observers has the following form:

```
var variable name: type = expression {
    willSet(setter name) {
        statements
    }
    didSet(setter name) {
        statements
    }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, the observers are referred to as *stored variable observers*. When it is declared in the context of a class or structure declaration, the observers are referred to as *property observers*.

You can add property observers to any stored property. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass, as described in Overriding Property Observers.

The initializer *expression* is optional in the context of a class or structure declaration, but required elsewhere. The *type* annotation is optional when the type can be inferred from the initializer *expression*.

The `willSet` and `didSet` observers provide a way to observe (and to respond appropriately) when the value of a variable or property is being set. The observers are not called when the variable or property is first initialized. Instead, they are called only when the value is set outside of an initialization context.

A `willSet` observer is called just before the value of the variable or property is set. The new value is passed to the `willSet` observer as a constant, and therefore it can't be changed in the implementation of the `willSet` clause. The `didSet` observer is called immediately after the new value is set. In contrast to the `willSet` observer, the old value of the variable or property is passed to the `didSet` observer in case you still need access to it. That said, if you assign a value to a variable or property within its own `didSet` observer clause, that new value that you assign will replace the one that was just set and passed to the `willSet` observer.

The *setter name* and enclosing parentheses in the `willSet` and `didSet` clauses are optional. If you provide setter names, they are used as the parameter names to the `willSet` and `didSet` observers. If you do not provide setter names, the default parameter name to the `willSet` observer is `newValue` and the default parameter name to the `didSet` observer is `oldValue`.

The `didSet` clause is optional when you provide a `willSet` clause. Likewise, the `willSet` clause is optional when you provide a `didSet` clause.

For more information and to see an example of how to use property observers, see Property Observers.

Type Variable Properties

To declare a type variable property, mark the declaration with the `static` declaration modifier. Classes can mark type computed properties with the `class` declaration modifier instead to allow subclasses to override the superclass's implementation. Type properties are discussed in Type Properties.

GRAMMAR OF A VARIABLE DECLARATION

variable-declaration → variable-declaration-head pattern-initializer-list
variable-declaration → variable-declaration-head variable-name type-annotation code-block
variable-declaration → variable-declaration-head variable-name type-annotation getter-setter-block
variable-declaration → variable-declaration-head variable-name type-annotation getter-setter-keyword-block
variable-declaration → variable-declaration-head variable-name initializer willSet-didSet-block
variable-declaration → variable-declaration-head variable-name type-annotation initializer$_{opt}$ willSet-didSet-block

variable-declaration-head → attributes$_{opt}$ declaration-modifiers$_{opt}$ **var**

variable-name → identifier

getter-setter-block → code-block
getter-setter-block → **{** getter-clause setter-clause$_{opt}$ **}**
getter-setter-block → **{** setter-clause getter-clause **}**
getter-clause → attributes$_{opt}$ mutation-modifier$_{opt}$ **get** code-block
setter-clause → attributes$_{opt}$ mutation-modifier$_{opt}$ **set** setter-name$_{opt}$ code-block
setter-name → **(** identifier **)**

getter-setter-keyword-block → **{** getter-keyword-clause setter-keyword-clause$_{opt}$ **}**
getter-setter-keyword-block → **{** setter-keyword-clause getter-keyword-clause **}**
getter-keyword-clause → attributes$_{opt}$ mutation-modifier$_{opt}$ **get**
setter-keyword-clause → attributes$_{opt}$ mutation-modifier$_{opt}$ **set**

willSet-didSet-block → **{** willSet-clause didSet-clause$_{opt}$ **}**
willSet-didSet-block → **{** didSet-clause willSet-clause$_{opt}$ **}**
willSet-clause → attributes$_{opt}$ **willSet** setter-name$_{opt}$ code-block
didSet-clause → attributes$_{opt}$ **didSet** setter-name$_{opt}$ code-block

Type Alias Declaration

A *type alias declaration* introduces a named alias of an existing type into your program. Type alias declarations are declared using the `typealias` keyword and have the following form:

```
typealias name = existing type
```

After a type alias is declared, the aliased *name* can be used instead of the *existing type* everywhere in your program. The *existing type* can be a named type or a compound type. Type aliases do not create new types; they simply allow a name to refer to an existing type.

A type alias declaration can use generic parameters to give a name to an existing generic type. The type alias can provide concrete types for some or all of the generic parameters of the existing type. For example:

```
typealias StringDictionary<Value> = Dictionary<String,
  Value>

```

```
// The following dictionaries have the same type.
var dictionary1: StringDictionary<Int> = [:]
var dictionary2: Dictionary<String, Int> = [:]
```

When a type alias is declared with generic parameters, the constraints on those parameters must match exactly the constraints on the existing type's generic parameters. For example:

```
typealias DictionaryOfInts<Key: Hashable> = Dictionary<Key,
  Int>
```

Because the type alias and the existing type can be used interchangeably, the type alias can't introduce additional generic constraints.

A type alias can forward an existing type's generic parameters by omitting all generic parameters from the declaration. For example, the `Diccionario` type alias declared here has the same generic parameters and constraints as `Dictionary`.

```
typealias Diccionario = Dictionary
```

Inside a protocol declaration, a type alias can give a shorter and more convenient name to a type that is used frequently. For example:

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    typealias Element = Iterator.Element
}

func sum<T: Sequence>(_ sequence: T) -> Int where T.Element
  == Int {
    // ...
}
```

Without this type alias, the `sum` function would have to refer to the associated type as `T.Iterator.Element` instead of `T.Element`.

See also Protocol Associated Type Declaration.

GRAMMAR OF A TYPE ALIAS DECLARATION

> *typealias-declaration* → attributes$_{opt}$ access-level-modifier$_{opt}$ **typealias** typealias-name generic-parameter-clause$_{opt}$ typealias-assignment
> *typealias-name* → identifier
> *typealias-assignment* → **=** type

Function Declaration

A *function declaration* introduces a function or method into your program. A function declared in the context of class, structure, enumeration, or protocol is referred to as a *method*. Function declarations are declared using the `func` keyword and have the following form:

```
func function name(parameters) -> return type {
    statements
}
```

If the function has a return type of `Void`, the return type can be omitted as follows:

```
func function name(parameters) {
    statements
}
```

The type of each parameter must be included—it can't be inferred. If you write `inout` in front of a parameter's type, the parameter can be modified inside the scope of the function. In-out parameters are discussed in detail in In-Out Parameters, below.

A function declaration whose *statements* include only a single expression is understood to return the value of that expression.

Functions can return multiple values using a tuple type as the return type of the function.

A function definition can appear inside another function declaration. This kind of function is known as a *nested function*.

A nested function is nonescaping if it captures a value that is guaranteed to never escape—such as an in-out parameter—or passed as a nonescaping function argument. Otherwise, the nested function is an escaping function.

For a discussion of nested functions, see Nested Functions.

Parameter Names

Function parameters are a comma-separated list where each parameter has one of several forms. The order of arguments in a function call must match the order of parameters in the function's declaration. The simplest entry in a parameter list has the following form:

```
parameter name: parameter type
```

A parameter has a name, which is used within the function body, as well as an argument label, which is used when calling the function or method. By default, parameter names are also used as argument labels. For example:

```
func f(x: Int, y: Int) -> Int { return x + y }
f(x: 1, y: 2) // both x and y are labeled
```

You can override the default behavior for argument labels with one of the following forms:

```
argument label parameter name: parameter type
_ parameter name: parameter type
```

A name before the parameter name gives the parameter an explicit argument label, which can be different from the parameter name. The corresponding argument must use the given argument label in function or method calls.

An underscore (_) before a parameter name suppresses the argument label. The corresponding argument must have no label in function or method calls.

```
func repeatGreeting(_ greeting: String, count n: Int) { /*
  Greet n times */ }
repeatGreeting("Hello, world!", count: 2) //  count is
  labeled, greeting is not
```

In-Out Parameters

In-out parameters are passed as follows:

1. When the function is called, the value of the argument is copied.
2. In the body of the function, the copy is modified.
3. When the function returns, the copy's value is assigned to the original argument.

This behavior is known as *copy-in copy-out* or *call by value result*. For example, when a computed property or a property with observers is passed as an in-out parameter, its getter is called as part of the function call and its setter is called as part of the function return.

As an optimization, when the argument is a value stored at a physical address in memory, the same memory location is used both inside and outside the function body. The optimized behavior is known as *call by reference*; it satisfies all of the requirements of the copy-in copy-out model while removing the overhead of copying. Write your code using the model given by copy-in copy-out, without depending on the call-by-reference optimization, so that it behaves correctly with or without the optimization.

Within a function, don't access a value that was passed as an in-out argument, even if the original value is available in the current scope. Accessing the original is a simultaneous access of the value, which violates Swift's memory exclusivity guarantee. For the same reason, you can't pass the same value to multiple in-out parameters.

For more information about memory safety and memory exclusivity, see Memory Safety.

A closure or nested function that captures an in-out parameter must be nonescaping. If you need to capture an in-out parameter without mutating it or to observe changes made by other code, use a capture list to explicitly capture the parameter immutably.

```
func someFunction(a: inout Int) -> () -> Int {
    return { [a] in return a + 1 }
}
```

If you need to capture and mutate an in-out parameter, use an explicit local copy, such as in multithreaded code that ensures all mutation has finished before the function returns.

```
func multithreadedFunction(queue: DispatchQueue, x: inout
  Int) {
    // Make a local copy and manually copy it back.
    var localX = x
    defer { x = localX }

    // Operate on localX asynchronously, then wait before
  returning.
    queue.async { someMutatingOperation(&localX) }
    queue.sync {}
}
```

For more discussion and examples of in-out parameters, see In-Out Parameters.

Special Kinds of Parameters

Parameters can be ignored, take a variable number of values, and provide default values using the following forms:

```
_ : parameter type
parameter name: parameter type...
parameter name: parameter type = default argument value
```

An underscore (_) parameter is explicitly ignored and can't be accessed within the body of the function.

A parameter with a base type name followed immediately by three dots (`...`) is understood as a variadic parameter. A function can have at most one variadic parameter. A variadic parameter is treated as an array that contains elements of the base type name. For example, the variadic parameter `Int...` is treated as `[Int]`. For an example that uses a variadic parameter, see Variadic Parameters.

A parameter with an equals sign (=) and an expression after its type is understood to have a default value of the given expression. The given expression is evaluated when the function is called. If the parameter is omitted when calling the function, the default value is used instead.

```
func f(x: Int = 42) -> Int { return x }
f()       // Valid, uses default value
f(x: 7)   // Valid, uses the value provided
f(7)      // Invalid, missing argument label
```

Special Kinds of Methods

Methods on an enumeration or a structure that modify `self` must be marked with the `mutating` declaration modifier.

Methods that override a superclass method must be marked with the `override` declaration modifier. It's a compile-time error to override a method without the `override` modifier or to use the `override` modifier on a method that doesn't override a superclass method.

Methods associated with a type rather than an instance of a type must be marked with the `static` declaration modifier for enumerations and structures, or with either the `static` or `class` declaration modifier for classes. A class type method marked with the `class` declaration modifier can be overridden by a subclass implementation; a class type method marked with `class final` or `static` can't be overridden.

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:

```
func function name(parameters) throws -> return type {
    statements
}
```

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.

```
func someFunction(callback: () throws -> Void) rethrows {
    try callback()
}
```

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()`.

```
func alwaysThrows() throws {
    throw SomeError.error
}
func someFunction(callback: () throws -> Void) rethrows {
```

```
    do {
        try callback()
        try alwaysThrows()  // Invalid, alwaysThrows() isn't
  a throwing parameter
    } catch {
        throw AnotherError.error
    }
}
```

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.

Functions that Never Return

Swift defines a `Never` type, which indicates that a function or method doesn't return to its caller. Functions and methods with the `Never` return type are called *nonreturning*. Nonreturning functions and methods either cause an irrecoverable error or begin a sequence of work that continues indefinitely. This means that code that would otherwise run immediately after the call is never executed. Throwing and rethrowing functions can transfer program control to an appropriate `catch` block, even when they are nonreturning.

A nonreturning function or method can be called to conclude the `else` clause of a guard statement, as discussed in Guard Statement.

You can override a nonreturning method, but the new method must preserve its return type and nonreturning behavior.

> GRAMMAR OF A FUNCTION DECLARATION
>
> *function-declaration* → function-head function-name generic-parameter-clause$_{opt}$ function-signature generic-where-clause$_{opt}$ function-body$_{opt}$
>
> *function-head* → attributes$_{opt}$ declaration-modifiers$_{opt}$ **func**
> *function-name* → identifier | operator
>
> *function-signature* → parameter-clause **throws**$_{opt}$ function-result$_{opt}$
> *function-signature* → parameter-clause **rethrows** function-result$_{opt}$
> *function-result* → **->** attributes$_{opt}$ type

function-body → code-block

parameter-clause → **(** **)** | **(** parameter-list **)**
parameter-list → parameter | parameter **,** parameter-list
parameter → external-parameter-name$_{opt}$ local-parameter-name type-annotation default-argument-clause$_{opt}$
parameter → external-parameter-name$_{opt}$ local-parameter-name type-annotation
parameter → external-parameter-name$_{opt}$ local-parameter-name type-annotation **...**
external-parameter-name → identifier
local-parameter-name → identifier
default-argument-clause → **=** expression

Enumeration Declaration

An *enumeration declaration* introduces a named enumeration type into your program.

Enumeration declarations have two basic forms and are declared using the `enum` keyword. The body of an enumeration declared using either form contains zero or more values—called *enumeration cases*—and any number of declarations, including computed properties, instance methods, type methods, initializers, type aliases, and even other enumeration, structure, and class declarations. Enumeration declarations can't contain deinitializer or protocol declarations.

Enumeration types can adopt any number of protocols, but can't inherit from classes, structures, or other enumerations.

Unlike classes and structures, enumeration types do not have an implicitly provided default initializer; all initializers must be declared explicitly. Initializers can delegate to other initializers in the enumeration, but the initialization process is complete only after an initializer assigns one of the enumeration cases to `self`.

Like structures but unlike classes, enumerations are value types; instances of an enumeration are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of an enumeration type with an extension declaration, as discussed in Extension Declaration.

Enumerations with Cases of Any Type

The following form declares an enumeration type that contains enumeration cases of any type:

```
enum enumeration name: adopted protocols {
    case enumeration case 1
    case enumeration case 2(associated value types)
}
```

Enumerations declared in this form are sometimes called *discriminated unions* in other programming languages.

In this form, each case block consists of the `case` keyword followed by one or more enumeration cases, separated by commas. The name of each case must be unique. Each case can also specify that it stores values of a given type. These types are specified in the *associated value types* tuple, immediately following the name of the case.

Enumeration cases that store associated values can be used as functions that create instances of the enumeration with the specified associated values. And just like functions, you can get a reference to an enumeration case and apply it later in your code.

```
enum Number {
    case integer(Int)
    case real(Double)
}
let f = Number.integer
// f is a function of type (Int) -> Number

// Apply f to create an array of Number instances with
  integer values
let evenInts: [Number] = [0, 2, 4, 6].map(f)
```

For more information and to see examples of cases with associated value types, see Associated Values.

Enumerations with Indirection

Enumerations can have a recursive structure, that is, they can have cases with associated values that are instances of the enumeration type itself. However, instances of enumeration types have value semantics, which means they have a fixed layout in memory. To support recursion, the compiler must insert a layer of indirection.

To enable indirection for a particular enumeration case, mark it with the `indirect` declaration modifier. An indirect case must have an associated value.

```
enum Tree<T> {
    case empty
    indirect case node(value: T, left: Tree, right: Tree)
}
```

To enable indirection for all the cases of an enumeration that have an associated value, mark the entire enumeration with the `indirect` modifier—this is convenient when the enumeration contains many cases that would each need to be marked with the `indirect` modifier.

An enumeration that is marked with the `indirect` modifier can contain a mixture of cases that have associated values and cases those that don't. That said, it can't contain any cases that are also marked with the `indirect` modifier.

Enumerations with Cases of a Raw-Value Type

The following form declares an enumeration type that contains enumeration cases of the same basic type:

```
enum enumeration name: raw-value type,
    adopted protocols {
    case enumeration case 1 = raw value 1
    case enumeration case 2 = raw value 2
}
```

In this form, each case block consists of the `case` keyword, followed by one or more enumeration cases, separated by commas. Unlike the cases in the first form, each case has an underlying value, called a *raw value*, of the same basic type. The type of these values is specified in the *raw-value type* and must represent an integer, floating-point number, string, or single character. In particular, the *raw-value type* must conform to the `Equatable` protocol and one of the following protocols: `ExpressibleByIntegerLiteral` for integer literals, `ExpressibleByFloatLiteral` for floating-point literals, `ExpressibleByStringLiteral` for string literals that contain any number of characters, and `ExpressibleByUnicodeScalarLiteral` or `ExpressibleByExtendedGraphemeClusterLiteral` for string literals that contain only a single character. Each case must have a unique name and be assigned a unique raw value.

If the raw-value type is specified as `Int` and you don't assign a value to the cases explicitly, they are implicitly assigned the values `0`, `1`, `2`, and so on. Each unassigned case of type `Int` is implicitly assigned a raw value that is automatically incremented from the raw value of the previous case.

```
enum ExampleEnum: Int {
    case a, b, c = 5, d
}
```

In the above example, the raw value of `ExampleEnum.a` is `0` and the value of `ExampleEnum.b` is `1`. And because the value of `ExampleEnum.c` is explicitly set to `5`, the value of `ExampleEnum.d` is automatically incremented from `5` and is therefore `6`.

If the raw-value type is specified as `String` and you don't assign values to the cases explicitly, each unassigned case is implicitly assigned a string with the same text as the name of that case.

```
enum GamePlayMode: String {
    case cooperative, individual, competitive
}
```

In the above example, the raw value of `GamePlayMode.cooperative` is `"cooperative"`, the raw value of `GamePlayMode.individual` is `"individual"`, and the raw value of `GamePlayMode.competitive` is `"competitive"`.

Enumerations that have cases of a raw-value type implicitly conform to the `RawRepresentable` protocol, defined in the Swift standard library. As a result, they have a `rawValue` property and a failable initializer with the signature `init?(rawValue: RawValue)`. You can use the `rawValue` property to access the raw value of an enumeration case, as in `ExampleEnum.b.rawValue`. You can also use a raw value to find a corresponding case, if there is one, by calling the enumeration's failable initializer, as in `ExampleEnum(rawValue: 5)`, which returns an optional case. For more information and to see examples of cases with raw-value types, see Raw Values.

Accessing Enumeration Cases

To reference the case of an enumeration type, use dot (`.`) syntax, as in `EnumerationType.enumerationCase`. When the enumeration type can be inferred from context, you can omit it (the dot is still required), as described in Enumeration Syntax and Implicit Member Expression.

To check the values of enumeration cases, use a `switch` statement, as shown in Matching Enumeration Values with a Switch Statement. The enumeration type is pattern-matched against the enumeration case patterns in the case blocks of the `switch` statement, as described in Enumeration Case Pattern.

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → attributes$_{opt}$ access-level-modifier$_{opt}$ union-style-enum
enum-declaration → attributes$_{opt}$ access-level-modifier$_{opt}$ raw-value-style-enum

union-style-enum → **indirect**$_{opt}$ **enum** enum-name generic-parameter-clause$_{opt}$ type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ **{** union-style-enum-members$_{opt}$ **}**
union-style-enum-members → union-style-enum-member union-style-enum-members$_{opt}$
union-style-enum-member → declaration | union-style-enum-case-clause | compiler-control-statement
union-style-enum-case-clause → attributes$_{opt}$ **indirect**$_{opt}$ **case** union-style-enum-case-list
union-style-enum-case-list → union-style-enum-case | union-style-enum-case **,** union-style-enum-case-list
union-style-enum-case → enum-case-name tuple-type$_{opt}$
enum-name → identifier
enum-case-name → identifier

raw-value-style-enum → **enum** enum-name generic-parameter-clause$_{opt}$ type-inheritance-clause generic-where-clause$_{opt}$ **{** raw-value-style-enum-members **}**
raw-value-style-enum-members → raw-value-style-enum-member raw-value-style-enum-members$_{opt}$
raw-value-style-enum-member → declaration | raw-value-style-enum-case-clause | compiler-control-statement
raw-value-style-enum-case-clause → attributes$_{opt}$ **case** raw-value-style-enum-case-list
raw-value-style-enum-case-list → raw-value-style-enum-case | raw-value-style-enum-case **,** raw-value-style-enum-case-list
raw-value-style-enum-case → enum-case-name raw-value-assignment$_{opt}$
raw-value-assignment → **=** raw-value-literal
raw-value-literal → numeric-literal | static-string-literal | boolean-literal

Structure Declaration

A *structure declaration* introduces a named structure type into your program. Structure declarations are declared using the `struct` keyword and have the following form:

```
struct structure name: adopted protocols {
    declarations
}
```

The body of a structure contains zero or more *declarations*. These *declarations* can include both stored and computed properties, type properties, instance methods, type methods, initializers, subscripts, type aliases, and even other structure, class, and enumeration declarations. Structure declarations can't contain deinitializer or protocol declarations. For a discussion and several examples of structures that include various kinds of declarations, see Structures and Classes.

Structure types can adopt any number of protocols, but can't inherit from classes, enumerations, or other structures.

There are three ways to create an instance of a previously declared structure:

- Call one of the initializers declared within the structure, as described in Initializers.

- If no initializers are declared, call the structure's memberwise initializer, as described in Memberwise Initializers for Structure Types.
- If no initializers are declared, and all properties of the structure declaration were given initial values, call the structure's default initializer, as described in Default Initializers.

The process of initializing a structure's declared properties is described in Initialization.

Properties of a structure instance can be accessed using dot (`.`) syntax, as described in Accessing Properties.

Structures are value types; instances of a structure are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of a structure type with an extension declaration, as discussed in Extension Declaration.

> GRAMMAR OF A STRUCTURE DECLARATION
>
> *struct-declaration* → attributes$_{opt}$ access-level-modifier$_{opt}$ **struct** struct-name generic-parameter-clause$_{opt}$ type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ struct-body
> *struct-name* → identifier
> *struct-body* → **{** struct-members$_{opt}$ **}**
>
> *struct-members* → struct-member struct-members$_{opt}$
> *struct-member* → declaration | compiler-control-statement

Class Declaration

A *class declaration* introduces a named class type into your program. Class declarations are declared using the `class` keyword and have the following form:

```
class class name: superclass, adopted protocols {
    declarations
}
```

The body of a class contains zero or more *declarations*. These *declarations* can include both stored and computed properties, instance methods, type methods, initializers, a single deinitializer, subscripts, type aliases, and even other class, structure, and enumeration declarations. Class declarations can't contain protocol declarations. For a discussion and several examples of classes that include various kinds of declarations, see Structures and Classes.

A class type can inherit from only one parent class, its *superclass*, but can adopt any number of protocols. The *superclass* appears first after the *class name* and colon, followed by any *adopted protocols*. Generic classes can inherit from other generic and nongeneric classes, but a nongeneric class can inherit only from other nongeneric classes. When you write the name of a generic superclass class after the colon, you must include the full name of that generic class, including its generic parameter clause.

As discussed in Initializer Declaration, classes can have designated and convenience initializers. The designated initializer of a class must initialize all of the class's declared properties and it must do so before calling any of its superclass's designated initializers.

A class can override properties, methods, subscripts, and initializers of its superclass. Overridden properties, methods, subscripts, and designated initializers must be marked with the `override` declaration modifier.

To require that subclasses implement a superclass's initializer, mark the superclass's initializer with the `required` declaration modifier. The subclass's implementation of that initializer must also be marked with the `required` declaration modifier.

Although properties and methods declared in the *superclass* are inherited by the current class, designated initializers declared in the *superclass* are only inherited when the subclass meets the conditions described in Automatic Initializer Inheritance. Swift classes do not inherit from a universal base class.

There are two ways to create an instance of a previously declared class:

- Call one of the initializers declared within the class, as described in Initializers.
- If no initializers are declared, and all properties of the class declaration were given initial values, call the class's default initializer, as described in Default Initializers.

Access properties of a class instance with dot (`.`) syntax, as described in Accessing Properties.

Classes are reference types; instances of a class are referred to, rather than copied, when assigned to variables or constants, or when passed as arguments to a function call. For information about reference types, see Structures and Enumerations Are Value Types.

You can extend the behavior of a class type with an extension declaration, as discussed in Extension Declaration.

GRAMMAR OF A CLASS DECLARATION

class-declaration → attributes$_{opt}$ access-level-modifier$_{opt}$ **final**$_{opt}$ **class** class-name generic-parameter-clause$_{opt}$ type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ class-body

class-declaration → attributes$_{opt}$ **final** access-level-modifier$_{opt}$ **class** class-name generic-parameter-clause$_{opt}$ type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ class-body

class-name → identifier

class-body → **{** class-members$_{opt}$ **}**

class-members → class-member class-members$_{opt}$

class-member → declaration | compiler-control-statement

Protocol Declaration

A *protocol declaration* introduces a named protocol type into your program. Protocol declarations are declared at global scope using the `protocol` keyword and have the following form:

```
protocol protocol name: inherited protocols {
    protocol member declarations
}
```

The body of a protocol contains zero or more *protocol member declarations*, which describe the conformance requirements that any type adopting the protocol must fulfill. In particular, a protocol can declare that conforming types must implement certain properties, methods, initializers, and subscripts. Protocols can also declare special kinds of type aliases, called *associated types*, that can specify relationships among the various declarations of the protocol. Protocol declarations can't contain class, structure, enumeration, or other protocol declarations. The *protocol member declarations* are discussed in detail below.

Protocol types can inherit from any number of other protocols. When a protocol type inherits from other protocols, the set of requirements from those other protocols are aggregated, and any type that inherits from the current protocol must conform to all those requirements. For an example of how to use protocol inheritance, see Protocol Inheritance.

> NOTE
>
> You can also aggregate the conformance requirements of multiple protocols using protocol composition types, as described in Protocol Composition Type and Protocol Composition.

You can add protocol conformance to a previously declared type by adopting the protocol in an extension declaration of that type. In the extension, you must implement all of the adopted protocol's requirements. If the type already implements all of the requirements, you can leave the body of the extension declaration empty.

By default, types that conform to a protocol must implement all properties, methods, and subscripts declared in the protocol. That said, you can mark these protocol member declarations with the `optional` declaration modifier to specify that their implementation by a conforming type is optional. The `optional` modifier can be applied only to members that are marked with the `objc` attribute, and only to members of protocols that are marked with the `objc` attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements. For more information about how to use the `optional` declaration modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see Optional Protocol Requirements.

To restrict the adoption of a protocol to class types only, include the `AnyObject` protocol in the *inherited protocols* list after the colon. For example, the following protocol can be adopted only by class types:

```
protocol SomeProtocol: AnyObject {
    /* Protocol members go here */
}
```

Any protocol that inherits from a protocol that's marked with the `AnyObject` requirement can likewise be adopted only by class types.

> NOTE
>
> If a protocol is marked with the `objc` attribute, the `AnyObject` requirement is implicitly applied to that protocol; there's no need to mark the protocol with the `AnyObject` requirement explicitly.

Protocols are named types, and thus they can appear in all the same places in your code as other named types, as discussed in Protocols as Types. However, you can't construct an instance of a protocol, because protocols do not actually provide the implementations for the requirements they specify.

You can use protocols to declare which methods a delegate of a class or structure should implement, as described in Delegation.

> GRAMMAR OF A PROTOCOL DECLARATION
>
> *protocol-declaration* → attributes$_{opt}$ access-level-modifier$_{opt}$ **protocol** protocol-name type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ protocol-body
> *protocol-name* → identifier
> *protocol-body* → **{** protocol-members$_{opt}$ **}**
>
> *protocol-members* → protocol-member protocol-members$_{opt}$
> *protocol-member* → protocol-member-declaration | compiler-control-statement
>
> *protocol-member-declaration* → protocol-property-declaration
> *protocol-member-declaration* → protocol-method-declaration
> *protocol-member-declaration* → protocol-initializer-declaration
> *protocol-member-declaration* → protocol-subscript-declaration
> *protocol-member-declaration* → protocol-associated-type-declaration
> *protocol-member-declaration* → typealias-declaration

Protocol Property Declaration

Protocols declare that conforming types must implement a property by including a *protocol property declaration* in the body of the protocol declaration. Protocol property declarations have a special form of a variable declaration:

```
var property name: type { get set }
```

As with other protocol member declarations, these property declarations declare only the getter and setter requirements for types that conform to the protocol. As a result, you don't implement the getter or setter directly in the protocol in which it is declared.

The getter and setter requirements can be satisfied by a conforming type in a variety of ways. If a property declaration includes both the `get` and `set` keywords, a conforming type can implement it with a stored variable property or a computed property that is both readable and writeable (that is, one that implements both a getter and a setter). However, that property declaration can't be implemented as a constant property or a read-only computed property. If a property declaration includes only the `get` keyword, it can be implemented as any kind of property. For examples of conforming types that implement the property requirements of a protocol, see Property Requirements.

To declare a type property requirement in a protocol declaration, mark the property declaration with the `static` keyword. Structures and enumerations that conform to the protocol declare the property with the `static` keyword, and classes that conform to the protocol declare the property with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a type property requirement use the `static` keyword.

See also Variable Declaration.

> GRAMMAR OF A PROTOCOL PROPERTY DECLARATION
>
> *protocol-property-declaration* → variable-declaration-head variable-name type-annotation getter-setter-keyword-block

Protocol Method Declaration

Protocols declare that conforming types must implement a method by including a protocol method declaration in the body of the protocol declaration. Protocol method declarations have the same form as function declarations, with two exceptions: They don't include a function body, and you can't provide any default parameter values as part of the function declaration. For examples of conforming types that implement the method requirements of a protocol, see Method Requirements.

To declare a class or static method requirement in a protocol declaration, mark the method declaration with the `static` declaration modifier. Structures and enumerations that conform to the protocol declare the method with the `static` keyword, and classes that conform to the protocol declare the method with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a type method requirement use the `static` keyword.

See also Function Declaration.

> GRAMMAR OF A PROTOCOL METHOD DECLARATION
>
> *protocol-method-declaration* → function-head function-name generic-parameter-clause *opt* function-signature generic-where-clause *opt*

Protocol Initializer Declaration

Protocols declare that conforming types must implement an initializer by including a protocol initializer declaration in the body of the protocol declaration. Protocol initializer declarations have the same form as initializer declarations, except they don't include the initializer's body.

A conforming type can satisfy a nonfailable protocol initializer requirement by implementing a nonfailable initializer or an `init!` failable initializer. A conforming type can satisfy a failable protocol initializer requirement by implementing any kind of initializer.

When a class implements an initializer to satisfy a protocol's initializer requirement, the initializer must be marked with the `required` declaration modifier if the class is not already marked with the `final` declaration modifier.

See also Initializer Declaration.

> GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION
>
> *protocol-initializer-declaration* → initializer-head generic-parameter-clause$_{opt}$
> parameter-clause **throws**$_{opt}$ generic-where-clause$_{opt}$
> *protocol-initializer-declaration* → initializer-head generic-parameter-clause$_{opt}$
> parameter-clause **rethrows** generic-where-clause$_{opt}$

Protocol Subscript Declaration

Protocols declare that conforming types must implement a subscript by including a protocol subscript declaration in the body of the protocol declaration. Protocol subscript declarations have a special form of a subscript declaration:

```
subscript (parameters) -> return type { get set }
```

Subscript declarations only declare the minimum getter and setter implementation requirements for types that conform to the protocol. If the subscript declaration includes both the `get` and `set` keywords, a conforming type must implement both a getter and a setter clause. If the subscript declaration includes only the `get` keyword, a conforming type must implement *at least* a getter clause and optionally can implement a setter clause.

To declare a static subscript requirement in a protocol declaration, mark the subscript declaration with the `static` declaration modifier. Structures and enumerations that conform to the protocol declare the subscript with the `static` keyword, and classes that conform to the protocol declare the subscript with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a static subscript requirement use the `static` keyword.

See also Subscript Declaration.

> GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

> *protocol-subscript-declaration* → subscript-head subscript-result generic-where-clause *opt* getter-setter-keyword-block

Protocol Associated Type Declaration

Protocols declare associated types using the `associatedtype` keyword. An associated type provides an alias for a type that is used as part of a protocol's declaration. Associated types are similar to type parameters in generic parameter clauses, but they're associated with `Self` in the protocol in which they're declared. In that context, `Self` refers to the eventual type that conforms to the protocol. For more information and examples, see Associated Types.

You use a generic `where` clause in a protocol declaration to add constraints to an associated types inherited from another protocol, without redeclaring the associated types. For example, the declarations of `SubProtocol` below are equivalent:

```
protocol SomeProtocol {
    associatedtype SomeType
}

protocol SubProtocolA: SomeProtocol {
    // This syntax produces a warning.
    associatedtype SomeType: Equatable
}

// This syntax is preferred.
protocol SubProtocolB: SomeProtocol where SomeType:
  Equatable { }
```

See also Type Alias Declaration.

> GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION
>
> *protocol-associated-type-declaration* → attributes *opt* access-level-modifier *opt* **associatedtype** typealias-name type-inheritance-clause *opt* typealias-assignment *opt* generic-where-clause *opt*

Initializer Declaration

An *initializer declaration* introduces an initializer for a class, structure, or enumeration into your program. Initializer declarations are declared using the `init` keyword and have two basic forms.

Structure, enumeration, and class types can have any number of initializers, but the rules and associated behavior for class initializers are different. Unlike structures and enumerations, classes have two kinds of initializers: designated initializers and convenience initializers, as described in Initialization.

The following form declares initializers for structures, enumerations, and designated initializers of classes:

```
init(parameters) {
    statements
}
```

A designated initializer of a class initializes all of the class's properties directly. It can't call any other initializers of the same class, and if the class has a superclass, it must call one of the superclass's designated initializers. If the class inherits any properties from its superclass, one of the superclass's designated initializers must be called before any of these properties can be set or modified in the current class.

Designated initializers can be declared in the context of a class declaration only and therefore can't be added to a class using an extension declaration.

Initializers in structures and enumerations can call other declared initializers to delegate part or all of the initialization process.

To declare convenience initializers for a class, mark the initializer declaration with the `convenience` declaration modifier.

```
convenience init(parameters) {
    statements
}
```

Convenience initializers can delegate the initialization process to another convenience initializer or to one of the class's designated initializers. That said, the initialization processes must end with a call to a designated initializer that ultimately initializes the class's properties. Convenience initializers can't call a superclass's initializers.

You can mark designated and convenience initializers with the `required` declaration modifier to require that every subclass implement the initializer. A subclass's implementation of that initializer must also be marked with the `required` declaration modifier.

By default, initializers declared in a superclass are not inherited by subclasses. That said, if a subclass initializes all of its stored properties with default values and doesn't define any initializers of its own, it inherits all of the superclass's initializers. If the subclass overrides all of the superclass's designated initializers, it inherits the superclass's convenience initializers.

As with methods, properties, and subscripts, you need to mark overridden designated initializers with the `override` declaration modifier.

> NOTE
>
> If you mark an initializer with the `required` declaration modifier, you don't also mark the initializer with the `override` modifier when you override the required initializer in a subclass.

Just like functions and methods, initializers can throw or rethrow errors. And just like functions and methods, you use the `throws` or `rethrows` keyword after an initializer's parameters to indicate the appropriate behavior.

To see examples of initializers in various type declarations, see Initialization.

Failable Initializers

A *failable initializer* is a type of initializer that produces an optional instance or an implicitly unwrapped optional instance of the type the initializer is declared on. As a result, a failable initializer can return `nil` to indicate that initialization failed.

To declare a failable initializer that produces an optional instance, append a question mark to the `init` keyword in the initializer declaration (`init?`). To declare a failable initializer that produces an implicitly unwrapped optional instance, append an exclamation mark instead (`init!`). The example below shows an `init?` failable initializer that produces an optional instance of a structure.

```
struct SomeStruct {
    let property: String
    // produces an optional instance of 'SomeStruct'
    init?(input: String) {
        if input.isEmpty {
            // discard 'self' and return 'nil'
            return nil
        }
        property = input
    }
}
```

You call an `init?` failable initializer in the same way that you call a nonfailable initializer, except that you must deal with the optionality of the result.

```
if let actualInstance = SomeStruct(input: "Hello") {
    // do something with the instance of 'SomeStruct'
} else {
    // initialization of 'SomeStruct' failed and the
  initializer returned 'nil'
}
```

A failable initializer can return `nil` at any point in the implementation of the initializer's body.

A failable initializer can delegate to any kind of initializer. A nonfailable initializer can delegate to another nonfailable initializer or to an `init!` failable initializer. A nonfailable initializer can delegate to an `init?` failable initializer by force-unwrapping the result of the superclass's initializer—for example, by writing `super.init()!`.

Initialization failure propagates through initializer delegation. Specifically, if a failable initializer delegates to an initializer that fails and returns `nil`, then the initializer that delegated also fails and implicitly returns `nil`. If a nonfailable initializer delegates to an `init!` failable initializer that fails and returns `nil`, then a runtime error is raised (as if you used the `!` operator to unwrap an optional that has a `nil` value).

A failable designated initializer can be overridden in a subclass by any kind of designated initializer. A nonfailable designated initializer can be overridden in a subclass by a nonfailable designated initializer only.

For more information and to see examples of failable initializers, see Failable Initializers.

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → initializer-head generic-parameter-clause$_{opt}$ parameter-clause **throws**$_{opt}$ generic-where-clause$_{opt}$ initializer-body

initializer-declaration → initializer-head generic-parameter-clause$_{opt}$ parameter-clause **rethrows** generic-where-clause$_{opt}$ initializer-body

initializer-head → attributes$_{opt}$ declaration-modifiers$_{opt}$ **init**

initializer-head → attributes$_{opt}$ declaration-modifiers$_{opt}$ **init ?**

initializer-head → attributes$_{opt}$ declaration-modifiers$_{opt}$ **init !**

initializer-body → code-block

Deinitializer Declaration

A *deinitializer declaration* declares a deinitializer for a class type. Deinitializers take no parameters and have the following form:

```
deinit {
    statements
}
```

A deinitializer is called automatically when there are no longer any references to a class object, just before the class object is deallocated. A deinitializer can be declared only in the body of a class declaration—but not in an extension of a class—and each class can have at most one.

A subclass inherits its superclass's deinitializer, which is implicitly called just before the subclass object is deallocated. The subclass object is not deallocated until all deinitializers in its inheritance chain have finished executing.

Deinitializers are not called directly.

For an example of how to use a deinitializer in a class declaration, see Deinitialization.

> GRAMMAR OF A DEINITIALIZER DECLARATION
>
> *deinitializer-declaration* → attributes$_{opt}$ **deinit** code-block

Extension Declaration

An *extension declaration* allows you to extend the behavior of existing types. Extension declarations are declared using the `extension` keyword and have the following form:

```
extension type name where requirements {
    declarations
}
```

The body of an extension declaration contains zero or more *declarations*. These *declarations* can include computed properties, computed type properties, instance methods, type methods, initializers, subscript declarations, and even class, structure, and enumeration declarations. Extension declarations can't contain deinitializer or protocol declarations, stored properties, property observers, or other extension declarations. Declarations in a protocol extension can't be marked `final`. For a discussion and several examples of extensions that include various kinds of declarations, see Extensions.

If the *type name* is a class, structure, or enumeration type, the extension extends that type. If the *type name* is a protocol type, the extension extends all types that conform to that protocol.

Extension declarations that extend a generic type or a protocol with associated types can include *requirements*. If an instance of the extended type or of a type that conforms to the extended protocol satisfies the *requirements*, the instance gains the behavior specified in the declaration.

Extension declarations can contain initializer declarations. That said, if the type you're extending is defined in another module, an initializer declaration must delegate to an initializer already defined in that module to ensure members of that type are properly initialized.

Properties, methods, and initializers of an existing type can't be overridden in an extension of that type.

Extension declarations can add protocol conformance to an existing class, structure, or enumeration type by specifying *adopted protocols*:

```
extension type name: adopted protocols where
   requirements {
    declarations
}
```

Extension declarations can't add class inheritance to an existing class, and therefore you can specify only a list of protocols after the *type name* and colon.

Conditional Conformance

You can extend a generic type to conditionally conform to a protocol, so that instances of the type conform to the protocol only when certain requirements are met. You add conditional conformance to a protocol by including *requirements* in an extension declaration.

Overridden Requirements Aren't Used in Some Generic Contexts

In some generic contexts, types that get behavior from conditional conformance to a protocol don't always use the specialized implementations of that protocol's requirements. To illustrate this behavior, the following example defines two protocols and a generic type that conditionally conforms to both protocols.

```
protocol Loggable {
    func log()
}
extension Loggable {
    func log() {
        print(self)
    }
}

protocol TitledLoggable: Loggable {
    static var logTitle: String { get }
}
extension TitledLoggable {
    func log() {
        print("\(Self.logTitle): \(self)")
    }
}

struct Pair<T>: CustomStringConvertible {
    let first: T
    let second: T
    var description: String {
        return "(\(first), \(second))"
    }
}

extension Pair: Loggable where T: Loggable { }
extension Pair: TitledLoggable where T: TitledLoggable {
    static var logTitle: String {
        return "Pair of '\(T.logTitle)'"
    }
}

extension String: TitledLoggable {
    static var logTitle: String {
        return "String"
    }
}
```

The `Pair` structure conforms to `Loggable` and `TitledLoggable` whenever its generic type conforms to `Loggable` or `TitledLoggable`, respectively. In the example below, `oneAndTwo` is an instance of `Pair<String>`, which conforms to `TitledLoggable` because `String` conforms to `TitledLoggable`. When the `log()` method is called on `oneAndTwo` directly, the specialized version containing the title string is used.

```
let oneAndTwo = Pair(first: "one", second: "two")
oneAndTwo.log()
// Prints "Pair of 'String': (one, two)"
```

However, when `oneAndTwo` is used in a generic context or as an instance of the `Loggable` protocol, the specialized version isn't used. Swift picks which implementation of `log()` to call by consulting only the minimum requirements that `Pair` needs to conform to `Loggable`. For this reason, the default implementation provided by the `Loggable` protocol is used instead.

```
func doSomething<T: Loggable>(with x: T) {
    x.log()
}
doSomething(with: oneAndTwo)
// Prints "(one, two)"
```

When `log()` is called on the instance that's passed to `doSomething(_:)`, the customized title is omitted from the logged string.

Protocol Conformance Must Not Be Redundant

A concrete type can conform to a particular protocol only once. Swift marks redundant protocol conformances as an error. You're likely to encounter this kind of error in two kinds of situations. The first situation is when you explicitly conform to the same protocol multiple times, but with different requirements. The second situation is when you implicitly inherit from the same protocol multiple times. These situations are discussed in the sections below.

Resolving Explicit Redundancy

Multiple extensions on a concrete type can't add conformance to the same protocol, even if the extensions' requirements are mutually exclusive. This restriction is demonstrated in the example below. Two extension declarations attempt to add conditional conformance to the `Serializable` protocol, one for for arrays with `Int` elements, and one for arrays with `String` elements.

```
protocol Serializable {
    func serialize() -> Any
}

extension Array: Serializable where Element == Int {
    func serialize() -> Any {
        // implementation
    }
}
extension Array: Serializable where Element == String {
    func serialize() -> Any {
        // implementation
    }
}
// Error: redundant conformance of 'Array<Element>' to
  protocol 'Serializable'
```

If you need to add conditional conformance based on multiple concrete types, create a new protocol that each type can conform to and use that protocol as the requirement when declaring conditional conformance.

```
protocol SerializableInArray { }
extension Int: SerializableInArray { }
extension String: SerializableInArray { }

extension Array: Serializable where Element:
  SerializableInArray {
    func serialize() -> Any {
        // implementation
    }
}
```

Resolving Implicit Redundancy

When a concrete type conditionally conforms to a protocol, that type implicitly conforms to any parent protocols with the same requirements.

If you need a type to conditionally conform to two protocols that inherit from a single parent, explicitly declare conformance to the parent protocol. This avoids implicitly conforming to the parent protocol twice with different requirements.

The following example explicitly declares the conditional conformance of `Array` to `Loggable` to avoid a conflict when declaring its conditional conformance to both `TitledLoggable` and the new `MarkedLoggable` protocol.

```
protocol MarkedLoggable: Loggable {
    func markAndLog()
}

extension MarkedLoggable {
    func markAndLog() {
        print("----------")
        log()
    }
}

extension Array: Loggable where Element: Loggable { }
extension Array: TitledLoggable where Element:
  TitledLoggable {
    static var logTitle: String {
        return "Array of '\(Element.logTitle)'"
    }
}
extension Array: MarkedLoggable where Element:
  MarkedLoggable { }
```

Without the extension to explicitly declare conditional conformance to `Loggable`, the other `Array` extensions would implicitly create these declarations, resulting in an error:

```
extension Array: Loggable where Element: TitledLoggable { }
```

```
extension Array: Loggable where Element: MarkedLoggable { }
// Error: redundant conformance of 'Array<Element>' to
  protocol 'Loggable'
```

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → attributes$_{opt}$ access-level-modifier$_{opt}$ **extension** type-identifier type-inheritance-clause$_{opt}$ generic-where-clause$_{opt}$ extension-body
extension-body → **{** extension-members$_{opt}$ **}**

extension-members → extension-member extension-members$_{opt}$
extension-member → declaration | compiler-control-statement

Subscript Declaration

A *subscript* declaration allows you to add subscripting support for objects of a particular type and are typically used to provide a convenient syntax for accessing the elements in a collection, list, or sequence. Subscript declarations are declared using the `subscript` keyword and have the following form:

```
subscript (parameters) -> return type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

Subscript declarations can appear only in the context of a class, structure, enumeration, extension, or protocol declaration.

The *parameters* specify one or more indexes used to access elements of the corresponding type in a subscript expression (for example, the `i` in the expression `object[i]`). Although the indexes used to access the elements can be of any type, each parameter must include a type annotation to specify the type of each index. The *return type* specifies the type of the element being accessed.

As with computed properties, subscript declarations support reading and writing the value of the accessed elements. The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly. That said, if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses are optional. If you provide a setter name, it is used as the name of the parameter to the setter. If you do not provide a setter name, the default parameter name to the setter is `value`. The type of the parameter to the setter is the same as the *return type*.

You can overload a subscript declaration in the type in which it is declared, as long as the *parameters* or the *return type* differ from the one you're overloading. You can also override a subscript declaration inherited from a superclass. When you do so, you must mark the overridden subscript declaration with the `override` declaration modifier.

Subscript parameters follow the same rules as function parameters, with two exceptions. By default, the parameters used in subscripting don't have argument labels, unlike functions, methods, and initializers. However, you can provide explicit argument labels using the same syntax that functions, methods, and initializers use. In addition, subscripts can't have in-out parameters.

You can also declare subscripts in the context of a protocol declaration, as described in Protocol Subscript Declaration.

For more information about subscripting and to see examples of subscript declarations, see Subscripts.

Type Subscript Declarations

To declare a subscript that's exposed by the type, rather than by instances of the type, mark the subscript declaration with the `static` declaration modifier. Classes can mark type computed properties with the `class` declaration modifier instead to allow subclasses to override the superclass's implementation. In a class declaration, the `static` keyword has the same effect as marking the declaration with both the `class` and `final` declaration modifiers.

> GRAMMAR OF A SUBSCRIPT DECLARATION

subscript-declaration → subscript-head subscript-result generic-where-clause$_{opt}$ code-block
subscript-declaration → subscript-head subscript-result generic-where-clause$_{opt}$ getter-setter-block
subscript-declaration → subscript-head subscript-result generic-where-clause$_{opt}$ getter-setter-keyword-block
subscript-head → attributes$_{opt}$ declaration-modifiers$_{opt}$ **subscript** generic-parameter-clause$_{opt}$ parameter-clause
subscript-result → **->** attributes$_{opt}$ type

Operator Declaration

An *operator declaration* introduces a new infix, prefix, or postfix operator into your program and is declared using the `operator` keyword.

You can declare operators of three different fixities: infix, prefix, and postfix. The *fixity* of an operator specifies the relative position of an operator to its operands.

There are three basic forms of an operator declaration, one for each fixity. The fixity of the operator is specified by marking the operator declaration with the `infix`, `prefix`, or `postfix` declaration modifier before the `operator` keyword. In each form, the name of the operator can contain only the operator characters defined in Operators.

The following form declares a new infix operator:

```
infix operator operator name: precedence group
```

An *infix operator* is a binary operator that is written between its two operands, such as the familiar addition operator (`+`) in the expression `1 + 2`.

Infix operators can optionally specify a precedence group. If you omit the precedence group for an operator, Swift uses the default precedence group, `DefaultPrecedence`, which specifies a precedence just higher than `TernaryPrecedence`. For more information, see Precedence Group Declaration.

The following form declares a new prefix operator:

```
prefix operator operator name
```

A *prefix operator* is a unary operator that is written immediately before its operand, such as the prefix logical NOT operator (`!`) in the expression `!a`.

Prefix operators declarations don't specify a precedence level. Prefix operators are nonassociative.

The following form declares a new postfix operator:

```
postfix operator operator name
```

A *postfix operator* is a unary operator that is written immediately after its operand, such as the postfix forced-unwrap operator (`!`) in the expression `a!`.

As with prefix operators, postfix operator declarations don't specify a precedence level. Postfix operators are nonassociative.

After declaring a new operator, you implement it by declaring a static method that has the same name as the operator. The static method is a member of one of the types whose values the operator takes as an argument—for example, an operator that multiplies a `Double` by an `Int` is implemented as a static method on either the `Double` or `Int` structure. If you're implementing a prefix or postfix operator, you must also mark that method declaration with the corresponding `prefix` or `postfix` declaration modifier. To see an example of how to create and implement a new operator, see Custom Operators.

GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → prefix-operator-declaration | postfix-operator-declaration | infix-operator-declaration

prefix-operator-declaration → **prefix operator** operator
postfix-operator-declaration → **postfix operator** operator
infix-operator-declaration → **infix operator** operator infix-operator-group$_{opt}$

infix-operator-group → **:** precedence-group-name

Precedence Group Declaration

A *precedence group declaration* introduces a new grouping for infix operator precedence into your program. The precedence of an operator specifies how tightly the operator binds to its operands, in the absence of grouping parentheses.

A precedence group declaration has the following form:

```
precedencegroup precedence group name {
    higherThan: lower group names
    lowerThan: higher group names
    associativity: associativity
    assignment: assignment
}
```

The *lower group names* and *higher group names* lists specify the new precedence group's relation to existing precedence groups. The `lowerThan` precedence group attribute may only be used to refer to precedence groups declared outside of the current module. When two operators compete with each other for their operands, such as in the expression `2 + 3 * 5`, the operator with the higher relative precedence binds more tightly to its operands.

> NOTE
>
> Precedence groups related to each other using *lower group names* and *higher group names* must fit into a single relational hierarchy, but they *don't* have to form a linear hierarchy. This means it is possible to have precedence groups with undefined relative precedence. Operators from those precedence groups can't be used next to each other without grouping parentheses.

Swift defines numerous precedence groups to go along with the operators provided by the standard library. For example, the addition (`+`) and subtraction (`-`) operators belong to the `AdditionPrecedence` group, and the multiplication (`*`) and division (`/`) operators belong to the `MultiplicationPrecedence` group. For a complete list of precedence groups provided by the Swift standard library, see Operator Declarations.

The *associativity* of an operator specifies how a sequence of operators with the same precedence level are grouped together in the absence of grouping parentheses. You specify the associativity of an operator by writing one of the context-sensitive keywords `left`, `right`, or `none`—if your omit the associativity, the default is `none`. Operators that are left-associative group left-to-right. For example, the subtraction operator (–) is left-associative, so the expression `4 – 5 – 6` is grouped as `(4 – 5) – 6` and evaluates to `–7`. Operators that are right-associative group right-to-left, and operators that are specified with an associativity of `none` don't associate at all. Nonassociative operators of the same precedence level can't appear adjacent to each to other. For example, the `<` operator has an associativity of `none`, which means `1 < 2 < 3` is not a valid expression.

The *assignment* of a precedence group specifies the precedence of an operator when used in an operation that includes optional chaining. When set to `true`, an operator in the corresponding precedence group uses the same grouping rules during optional chaining as the assignment operators from the standard library. Otherwise, when set to `false` or omitted, operators in the precedence group follows the same optional chaining rules as operators that don't perform assignment.

GRAMMAR OF A PRECEDENCE GROUP DECLARATION

precedence-group-declaration → **precedencegroup** precedence-group-name **{** precedence-group-attributes*opt* **}**

precedence-group-attributes → precedence-group-attribute precedence-group-attributes*opt*
precedence-group-attribute → precedence-group-relation
precedence-group-attribute → precedence-group-assignment
precedence-group-attribute → precedence-group-associativity

precedence-group-relation → **higherThan :** precedence-group-names
precedence-group-relation → **lowerThan :** precedence-group-names

precedence-group-assignment → **assignment :** boolean-literal

precedence-group-associativity → **associativity : left**
precedence-group-associativity → **associativity : right**
precedence-group-associativity → **associativity : none**

precedence-group-names → precedence-group-name | precedence-group-name **,** precedence-group-names
precedence-group-name → identifier

Declaration Modifiers

Declaration modifiers are keywords or context-sensitive keywords that modify the behavior or meaning of a declaration. You specify a declaration modifier by writing the appropriate keyword or context-sensitive keyword between a declaration's attributes (if any) and the keyword that introduces the declaration.

`class`
: Apply this modifier to a member of a class to indicate that the member is a member of the class itself, rather than a member of instances of the class. Members of a superclass that have this modifier and don't have the `final` modifier can be overridden by subclasses.

`dynamic`
: Apply this modifier to any member of a class that can be represented by Objective-C. When you mark a member declaration with the `dynamic` modifier, access to that member is always dynamically dispatched using the Objective-C runtime. Access to that member is never inlined or devirtualized by the compiler.

  Because declarations marked with the `dynamic` modifier are dispatched using the Objective-C runtime, they must be marked with the `objc` attribute.

`final`
: Apply this modifier to a class or to a property, method, or subscript member of a class. It's applied to a class to indicate that the class can't be subclassed. It's applied to a property, method, or subscript of a class to indicate that a class member can't be overridden in any subclass. For an example of how to use the `final` attribute, see Preventing Overrides.

`lazy`
: Apply this modifier to a stored variable property of a class or structure to indicate that the property's initial value is calculated and stored at most once, when the property is first accessed. For an example of how to use the `lazy` modifier, see Lazy Stored Properties.

`optional`
: Apply this modifier to a protocol's property, method, or subscript members to indicate that a conforming type isn't required to implement those members.

You can apply the `optional` modifier only to protocols that are marked with the `objc` attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements. For more information about how to use the `optional` modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see Optional Protocol Requirements.

`required`
Apply this modifier to a designated or convenience initializer of a class to indicate that every subclass must implement that initializer. The subclass's implementation of that initializer must also be marked with the `required` modifier.

`static`
Apply this modifier to a member of a structure, class, enumeration, or protocol to indicate that the member is a member of the type, rather than a member of instances of that type. In the scope of a class declaration, writing the `static` modifier on a member declaration has the same effect as writing the `class` and `final` modifiers on that member declaration. However, constant type properties of a class are an exception: `static` has its normal, nonclass meaning there because you can't write `class` or `final` on those declarations.

`unowned`
Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, a runtime error is raised. Like a weak reference, the type of the property or value must be a class type; unlike a weak reference, the type is non-optional. For an example and more information about the `unowned` modifier, see Unowned References.

`unowned(safe)`
An explicit spelling of `unowned`.

`unowned(unsafe)`
Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, you'll access the memory at the location where the object used to be, which is a memory-unsafe operation. Like a weak reference, the type of the

property or value must be a class type; unlike a weak reference, the type is non-optional. For an example and more information about the `unowned` modifier, see Unowned References.

`weak`
Apply this modifier to a stored variable or stored variable property to indicate that the variable or property has a weak reference to the object stored as its value. The type of the variable or property must be an optional class type. If you access the variable or property after the object has been deallocated, its value is `nil`. For an example and more information about the `weak` modifier, see Weak References.

Access Control Levels

Swift provides five levels of access control: open, public, internal, file private, and private. You can mark a declaration with one of the access-level modifiers below to specify the declaration's access level. Access control is discussed in detail in Access Control.

`open`
Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the `open` access-level modifier can also be accessed and subclassed by code in a module that imports the module that contains that declaration.

`public`
Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the `public` access-level modifier can also be accessed (but not subclassed) by code in a module that imports the module that contains that declaration.

`internal`
Apply this modifier to a declaration to indicate the declaration can be accessed only by code in the same module as the declaration. By default, most declarations are implicitly marked with the `internal` access-level modifier.

`fileprivate`
Apply this modifier to a declaration to indicate the declaration can be accessed only by code in the same source file as the declaration.

`private`

Apply this modifier to a declaration to indicate the declaration can be accessed only by code within the declaration's immediate enclosing scope.

For the purpose of access control, extensions to the same type that are in the same file share an access-control scope. If the type they extend is also in the same file, they share the type's access-control scope. Private members declared in the type's declaration can be accessed from extensions, and private members declared in one extension can be accessed from other extensions and from the type's declaration.

Each access-level modifier above optionally accepts a single argument, which consists of the `set` keyword enclosed in parentheses (for example, `private(set)`). Use this form of an access-level modifier when you want to specify an access level for the setter of a variable or subscript that's less than or equal to the access level of the variable or subscript itself, as discussed in Getters and Setters.

GRAMMAR OF A DECLARATION MODIFIER

declaration-modifier → **class** | **convenience** | **dynamic** | **final** | **infix** | **lazy** | **optional** | **override** | **postfix** | **prefix** | **required** | **static** | **unowned** | **unowned (safe)** | **unowned (unsafe)** | **weak**
declaration-modifier → access-level-modifier
declaration-modifier → mutation-modifier
declaration-modifiers → declaration-modifier declaration-modifiers$_{opt}$

access-level-modifier → **private** | **private (set)**
access-level-modifier → **fileprivate** | **fileprivate (set)**
access-level-modifier → **internal** | **internal (set)**
access-level-modifier → **public** | **public (set)**
access-level-modifier → **open** | **open (set)**

mutation-modifier → **mutating** | **nonmutating**

‹ Statements Attributes ›

> Learn more about using Apple's beta software