Swift - Tuples

Swift 4 also introduces **Tuples** type, which are used to group multiple values in a single compound Value.

The values in a tuple can be of any type, and do not need to be of same type.

For example, ("Tutorials Point", 123) is a tuple with two values, one of string Type, and other is integer type. It is a legal command.

let ImplementationError = (501, "Not implemented") is an error when something on the server is not implemented, It returns two values. Error Code, and Description.

You can create tuples from as many values as you want and from any number of different data types.

Here’s the syntax of Tuple declaration −

var TupleName = (Value1, value2,… any number of values)

Here’s a Tuple declaration −

var error501 = (501, “Not implemented”)

You can access the values of tuple using the index numbers that start from 0.

Here’s an example of accessing tuple Values −

print(“The code is\(error501.0)”)

print(“The definition of error is\(error501.1)”)

You can name the variables of a tuple while declaring , and you can call them using their names

var error501 = (errorCode: 501, description: “Not Implemented”)

print(error501.errorCode) // prints 501.

Tuples are helpful in returning multiple values from a function. Like, a web application might return a tuple of type ("String", Int) to show whether the loading was successful or failed.

By returning different values in a tuple we can make decisions depending on different tuple types.

**Note** − Tuples are useful for temporary values and are not suited for complex data.

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

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

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

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

This chapter discusses the types defined in the Swift language itself and describes the type inference behavior of Swift.

GRAMMAR OF A TYPE

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

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

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

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

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

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

*type* → [implicitly-unwrapped-optional-type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_implicitly-unwrapped-optional-type)

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

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

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

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

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

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

## **Type Annotation**

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

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

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

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

GRAMMAR OF A TYPE ANNOTATION

*type-annotation* → **:** [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **inout***opt* [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

## **Type Identifier**

A type identifier refers to either a named type or a type alias of a named or compound type.

Most of the time, a type identifier directly refers to a named type with the same name as the identifier. For example, Int is a type identifier that directly refers to the named type Int, and the type identifier Dictionary<String, Int> directly refers to the named type Dictionary<String, Int>.

There are two cases in which a type identifier doesn’t refer to a type with the same name. In the first case, a type identifier refers to a type alias of a named or compound type. For instance, in the example below, the use of Point in the type annotation refers to the tuple type (Int, Int).

1. typealias Point = (Int, Int)
2. let origin: Point = (0, 0)

In the second case, a type identifier uses dot (.) syntax to refer to named types declared in other modules or nested within other types. For example, the type identifier in the following code references the named type MyType that’s declared in the ExampleModule module.

1. var someValue: ExampleModule.MyType

GRAMMAR OF A TYPE IDENTIFIER

*type-identifier* → [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) [generic-argument-clause](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-argument-clause) *opt* | [type-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-name) [generic-argument-clause](https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html#grammar_generic-argument-clause) *opt* **.** [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier)

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

## **Tuple Type**

A tuple type is a comma-separated list of types, enclosed in parentheses.

You can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values. You can also name the elements of a tuple type and use those names to refer to the values of the individual elements. An element name consists of an identifier followed immediately by a colon (:). For an example that demonstrates both of these features, see [Functions with Multiple Return Values](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID164).

When an element of a tuple type has a name, that name is part of the type.

1. var someTuple = (top: 10, bottom: 12) // someTuple is of type (top: Int, bottom: Int)
2. someTuple = (top: 4, bottom: 42) // OK: names match
3. someTuple = (9, 99) // OK: names are inferred
4. someTuple = (left: 5, right: 5) // Error: names don't match

All tuple types contain two or more types, except for Void which is a type alias for the empty tuple type, ().

GRAMMAR OF A TUPLE TYPE

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

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

*tuple-type-element* → [element-name](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_element-name) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation) | [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

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

## **Function Type**

A function type represents the type of a function, method, or closure and consists of a parameter and return type separated by an arrow (->):

1. (parameter type) -> return type

The parameter type is comma-separated list of types. Because the return type can be a tuple type, function types support functions and methods that return multiple values.

A parameter of the function type () -> T (where T is any type) can apply the autoclosure attribute to implicitly create a closure at its call sites. This provides a syntactically convenient way to defer the evaluation of an expression without needing to write an explicit closure when you call the function. For an example of an autoclosure function type parameter, see [Autoclosures](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID543).

A function type can have variadic parameters in its parameter type. Syntactically, a variadic parameter consists of a base type name followed immediately by three dots (...), as in Int.... A variadic parameter is treated as an array that contains elements of the base type name. For instance, the variadic parameter Int... is treated as [Int]. For an example that uses a variadic parameter, see [Variadic Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID171).

To specify an in-out parameter, prefix the parameter type with the inout keyword. You can’t mark a variadic parameter or a return type with the inout keyword. In-out parameters are discussed in [In-Out Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID173).

If a function type has only one parameter and that parameter’s type is a tuple type, then the tuple type must be parenthesized when writing the function’s type. For example, ((Int, Int)) -> Void is the type of a function that takes a single parameter of the tuple type (Int, Int) and doesn’t return any value. In contrast, without parentheses, (Int, Int) -> Void is the type of a function that takes two Int parameters and doesn’t return any value. Likewise, because Void is a type alias for (), the function type (Void) -> Void is the same as (()) -> ()—a function that takes a single argument that’s an empty tuple. These types aren’t the same as () -> ()—a function that takes no arguments.

Argument names in functions and methods aren’t part of the corresponding function type. For example:

1. func someFunction(left: Int, right: Int) {}
2. func anotherFunction(left: Int, right: Int) {}
3. func functionWithDifferentLabels(top: Int, bottom: Int) {}
4. var f = someFunction // The type of f is (Int, Int) -> Void, not (left: Int, right: Int) -> Void.
5. f = anotherFunction // OK
6. f = functionWithDifferentLabels // OK
7. func functionWithDifferentArgumentTypes(left: Int, right: String) {}
8. f = functionWithDifferentArgumentTypes // Error
9. func functionWithDifferentNumberOfArguments(left: Int, right: Int, top: Int) {}
10. f = functionWithDifferentNumberOfArguments // Error

Because argument labels aren’t part of a function’s type, you omit them when writing a function type.

1. var operation: (lhs: Int, rhs: Int) -> Int // Error
2. var operation: (\_ lhs: Int, \_ rhs: Int) -> Int // OK
3. var operation: (Int, Int) -> Int // OK

If a function type includes more than a single arrow (->), the function types are grouped from right to left. For example, the function type (Int) -> (Int) -> Int is understood as (Int) -> ((Int) -> Int)—that is, a function that takes an Int and returns another function that takes and returns an Int.

Function types that can throw or rethrow an error must be marked with the throws keyword. 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. Throwing and rethrowing functions are described in [Throwing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID530) and [Rethrowing Functions and Methods](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID531).

### **Restrictions for Nonescaping Closures**

A parameter that’s a nonescaping function can’t be stored in a property, variable, or constant of type Any, because that might allow the value to escape.

A parameter that’s a nonescaping function can’t be passed as an argument to another nonescaping function parameter. This restriction helps Swift perform more of its checks for conflicting access to memory at compile time instead of at runtime. For example:

1. let external: (() -> Void) -> Void = { \_ in () }
2. func takesTwoFunctions(first: (() -> Void) -> Void, second: (() -> Void) -> Void) {
3. first { first {} } // Error
4. second { second {} } // Error
5. first { second {} } // Error
6. second { first {} } // Error
7. first { external {} } // OK
8. external { first {} } // OK
9. }

In the code above, both of the parameters to takesTwoFunctions(first:second:) are functions. Neither parameter is marked @escaping, so they’re both nonescaping as a result.

The four function calls marked “Error” in the example above cause compiler errors. Because the first and second parameters are nonescaping functions, they can’t be passed as arguments to another nonescaping function parameter. In contrast, the two function calls marked “OK” don’t cause a compiler error. These function calls don’t violate the restriction because external isn’t one of the parameters of takesTwoFunctions(first:second:).

If you need to avoid this restriction, mark one of the parameters as escaping, or temporarily convert one of the nonescaping function parameters to an escaping function by using the withoutActuallyEscaping(\_:do:) function. For information about avoiding conflicting access to memory, see [Memory Safety](https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html).

GRAMMAR OF A FUNCTION TYPE

*function-type* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* [function-type-argument-clause](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_function-type-argument-clause) **throws***opt* **->** [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type)

*function-type-argument-clause* → **(** **)**

*function-type-argument-clause* → **(** [function-type-argument-list](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_function-type-argument-list) **...***opt* **)**

*function-type-argument-list* → [function-type-argument](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_function-type-argument) | [function-type-argument](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_function-type-argument) **,** [function-type-argument-list](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_function-type-argument-list)

*function-type-argument* → [attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#grammar_attributes) *opt* **inout***opt* [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) | [argument-label](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_argument-label) [type-annotation](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-annotation)

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

## **Array Type**

The Swift language provides the following syntactic sugar for the Swift standard library Array<Element> type:

1. [type]

In other words, the following two declarations are equivalent:

1. let someArray: Array<String> = ["Alex", "Brian", "Dave"]
2. let someArray: [String] = ["Alex", "Brian", "Dave"]

In both cases, the constant someArray is declared as an array of strings. The elements of an array can be accessed through subscripting by specifying a valid index value in square brackets: someArray[0] refers to the element at index 0, "Alex".

You can create multidimensional arrays by nesting pairs of square brackets, where the name of the base type of the elements is contained in the innermost pair of square brackets. For example, you can create a three-dimensional array of integers using three sets of square brackets:

1. var array3D: [[[Int]]] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

When accessing the elements in a multidimensional array, the left-most subscript index refers to the element at that index in the outermost array. The next subscript index to the right refers to the element at that index in the array that’s nested one level in. And so on. This means that in the example above, array3D[0] refers to [[1, 2], [3, 4]], array3D[0][1] refers to [3, 4], and array3D[0][1][1] refers to the value 4.

For a detailed discussion of the Swift standard library Array type, see [Arrays](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID107).

GRAMMAR OF AN ARRAY TYPE

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

## **Dictionary Type**

The Swift language provides the following syntactic sugar for the Swift standard library Dictionary<Key, Value> type:

1. [key type: value type]

In other words, the following two declarations are equivalent:

1. let someDictionary: [String: Int] = ["Alex": 31, "Paul": 39]
2. let someDictionary: Dictionary<String, Int> = ["Alex": 31, "Paul": 39]

In both cases, the constant someDictionary is declared as a dictionary with strings as keys and integers as values.

The values of a dictionary can be accessed through subscripting by specifying the corresponding key in square brackets: someDictionary["Alex"] refers to the value associated with the key "Alex". The subscript returns an optional value of the dictionary’s value type. If the specified key isn’t contained in the dictionary, the subscript returns nil.

The key type of a dictionary must conform to the Swift standard library Hashable protocol.

For a detailed discussion of the Swift standard library Dictionary type, see [Dictionaries](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID113).

GRAMMAR OF A DICTIONARY TYPE

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

## **Optional Type**

The Swift language defines the postfix ? as syntactic sugar for the named type Optional<Wrapped>, which is defined in the Swift standard library. In other words, the following two declarations are equivalent:

1. var optionalInteger: Int?
2. var optionalInteger: Optional<Int>

In both cases, the variable optionalInteger is declared to have the type of an optional integer. Note that no whitespace may appear between the type and the ?.

The type Optional<Wrapped> is an enumeration with two cases, none and some(Wrapped), which are used to represent values that may or may not be present. Any type can be explicitly declared to be (or implicitly converted to) an optional type. If you don’t provide an initial value when you declare an optional variable or property, its value automatically defaults to nil.

If an instance of an optional type contains a value, you can access that value using the postfix operator !, as shown below:

1. optionalInteger = 42
2. optionalInteger! // 42

Using the ! operator to unwrap an optional that has a value of nil results in a runtime error.

You can also use optional chaining and optional binding to conditionally perform an operation on an optional expression. If the value is nil, no operation is performed and therefore no runtime error is produced.

For more information and to see examples that show how to use optional types, see [Optionals](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID330).

GRAMMAR OF AN OPTIONAL TYPE

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

## **Implicitly Unwrapped Optional Type**

The Swift language defines the postfix ! as syntactic sugar for the named type Optional<Wrapped>, which is defined in the Swift standard library, with the additional behavior that it’s automatically unwrapped when it’s accessed. If you try to use an implicitly unwrapped optional that has a value of nil, you’ll get a runtime error. With the exception of the implicit unwrapping behavior, the following two declarations are equivalent:

1. var implicitlyUnwrappedString: String!
2. var explicitlyUnwrappedString: Optional<String>

Note that no whitespace may appear between the type and the !.

Because implicit unwrapping changes the meaning of the declaration that contains that type, optional types that are nested inside a tuple type or a generic type—such as the element types of a dictionary or array—can’t be marked as implicitly unwrapped. For example:

1. let tupleOfImplicitlyUnwrappedElements: (Int!, Int!) // Error
2. let implicitlyUnwrappedTuple: (Int, Int)! // OK
3. let arrayOfImplicitlyUnwrappedElements: [Int!] // Error
4. let implicitlyUnwrappedArray: [Int]! // OK

Because implicitly unwrapped optionals have the same Optional<Wrapped> type as optional values, you can use implicitly unwrapped optionals in all the same places in your code that you can use optionals. For example, you can assign values of implicitly unwrapped optionals to variables, constants, and properties of optionals, and vice versa.

As with optionals, if you don’t provide an initial value when you declare an implicitly unwrapped optional variable or property, its value automatically defaults to nil.

Use optional chaining to conditionally perform an operation on an implicitly unwrapped optional expression. If the value is nil, no operation is performed and therefore no runtime error is produced.

For more information about implicitly unwrapped optional types, see [Implicitly Unwrapped Optionals](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID334).

GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE

*implicitly-unwrapped-optional-type* → [type](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type) **!**

## **Protocol Composition Type**

A protocol composition type defines a type that conforms to each protocol in a list of specified protocols, or a type that’s a subclass of a given class and conforms to each protocol in a list of specified protocols. Protocol composition types may be used only when specifying a type in type annotations, in generic parameter clauses, and in generic where clauses.

Protocol composition types have the following form:

1. Protocol 1 & Protocol 2

A protocol composition type allows you to specify a value whose type conforms to the requirements of multiple protocols without explicitly defining a new, named protocol that inherits from each protocol you want the type to conform to. For example, you can use the protocol composition type ProtocolA & ProtocolB & ProtocolC instead of declaring a new protocol that inherits from ProtocolA, ProtocolB, and ProtocolC. Likewise, you can use SuperClass & ProtocolA instead of declaring a new protocol that’s a subclass of SuperClass and conforms to ProtocolA.

Each item in a protocol composition list is one of the following; the list can contain at most one class:

* The name of a class
* The name of a protocol
* A type alias whose underlying type is a protocol composition type, a protocol, or a class.

When a protocol composition type contains type aliases, it’s possible for the same protocol to appear more than once in the definitions—duplicates are ignored. For example, the definition of PQR in the code below is equivalent to P & Q & R.

1. typealias PQ = P & Q
2. typealias PQR = PQ & Q & R

GRAMMAR OF A PROTOCOL COMPOSITION TYPE

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

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

## **Opaque Type**

An opaque type defines a type that conforms to a protocol or protocol composition, without specifying the underlying concrete type.

Opaque types appear as the return type of a function or subscript, or the type of a property. Opaque types can’t appear as part of a tuple type or a generic type, such as the element type of an array or the wrapped type of an optional.

Opaque types have the following form:

1. some constraint

The constraint is a class type, protocol type, protocol composition type, or Any. A value can be used as an instance of the opaque type only if it’s an instance of a type that conforms to the listed protocol or protocol composition, or inherits from the listed class. Code that interacts with an opaque value can use the value only in ways that are part of the interface defined by the constraint.

Protocol declarations can’t include opaque types. Classes can’t use an opaque type as the return type of a nonfinal method.

A function that uses an opaque type as its return type must return values that share a single underlying type. The return type can include types that are part of the function’s generic type parameters. For example, a function someFunction<T>() could return a value of type T or Dictionary<String, T>.

GRAMMAR OF AN OPAQUE TYPE

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

## **Metatype Type**

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

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

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

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

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

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

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

GRAMMAR OF A METATYPE TYPE

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

## **Any Type**

The Any type can contain values from all other types. Any can be used as the concrete type for an instance of any of the following types:

* A class, structure, or enumeration
* A metatype, such as Int.self
* A tuple with any types of components
* A closure or function type

1. let mixed: [Any] = ["one", 2, true, (4, 5.3), { () -> Int in return 6 }]

When you use Any as a concrete type for an instance, you need to cast the instance to a known type before you can access its properties or methods. Instances with a concrete type of Any maintain their original dynamic type and can be cast to that type using one of the type-cast operators—as, as?, or as!. For example, use as? to conditionally downcast the first object in a heterogeneous array to a String as follows:

1. if let first = mixed.first as? String {
2. print("The first item, '\(first)', is a string.")
3. }
4. // Prints "The first item, 'one', is a string."

For more information about casting, see [Type Casting](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html).

The AnyObject protocol is similar to the Any type. All classes implicitly conform to AnyObject. Unlike Any, which is defined by the language, AnyObject is defined by the Swift standard library. For more information, see [Class-Only Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID281) and [AnyObject](https://developer.apple.com/documentation/swift/anyobject).

GRAMMAR OF AN ANY TYPE

*any-type* → **Any**

## **Self Type**

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

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

In a structure, class, or enumeration declaration, the Self type refers to the type introduced by the declaration. Inside the declaration for a member of a type, the Self type refers to that type. In the members of a class declaration, Self can appear only as follows:

* As the return type of a method
* As the return type of a read-only subscript
* As the type of a read-only computed property
* In the body of a method

For example, the code below shows an instance method f whose return type is Self.

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

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

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

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

GRAMMAR OF A SELF TYPE

*self-type* → **Self**

## **Type Inheritance Clause**

A type inheritance clause is used to specify which class a named type inherits from and which protocols a named type conforms to. A type inheritance clause begins with a colon (:), followed by a list of type identifiers.

Class types can inherit from a single superclass and conform to any number of protocols. When defining a class, the name of the superclass must appear first in the list of type identifiers, followed by any number of protocols the class must conform to. If the class doesn’t inherit from another class, the list can begin with a protocol instead. For an extended discussion and several examples of class inheritance, see [Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html).

Other named types can only inherit from or conform to a list of protocols. 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 together, and any type that inherits from the current protocol must conform to all of those requirements.

A type inheritance clause in an enumeration definition can be either a list of protocols, or in the case of an enumeration that assigns raw values to its cases, a single, named type that specifies the type of those raw values. For an example of an enumeration definition that uses a type inheritance clause to specify the type of its raw values, see [Raw Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID149).

GRAMMAR OF A TYPE INHERITANCE CLAUSE

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

*type-inheritance-list* → [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) | [type-identifier](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-identifier) **,** [type-inheritance-list](https://docs.swift.org/swift-book/ReferenceManual/Types.html#grammar_type-inheritance-list)

## **Type Inference**

Swift uses type inference extensively, allowing you to omit the type or part of the type of many variables and expressions in your code. For example, instead of writing var x: Int = 0, you can write var x = 0, omitting the type completely—the compiler correctly infers that x names a value of type Int. Similarly, you can omit part of a type when the full type can be inferred from context. For example, if you write let dict: Dictionary = ["A": 1], the compiler infers that dict has the type Dictionary<String, Int>.

In both of the examples above, the type information is passed up from the leaves of the expression tree to its root. That is, the type of x in var x: Int = 0 is inferred by first checking the type of 0 and then passing this type information up to the root (the variable x).

In Swift, type information can also flow in the opposite direction—from the root down to the leaves. In the following example, for instance, the explicit type annotation (: Float) on the constant eFloat causes the numeric literal 2.71828 to have an inferred type of Float instead of Double.

1. let e = 2.71828 // The type of e is inferred to be Double.
2. let eFloat: Float = 2.71828 // The type of eFloat is Float.

Type inference in Swift operates at the level of a single expression or statement. This means that all of the information needed to infer an omitted type or part of a type in an expression must be accessible from type-checking the expression or one of its subexpressions.