ON THIS PAGE

Swift

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you've been using generics throughout the *Language Guide*, even if you didn't realize it. For example, Swift's `Array` and `Dictionary` types are both generic collections. You can create an array that holds `Int` values, or an array that holds `String` values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

The Problem That Generics Solve

Here's a standard, nongeneric function called `swapTwoInts(_:_:)`, which swaps two `Int` values:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

This function makes use of in-out parameters to swap the values of `a` and `b`, as described in In-Out Parameters.

The `swapTwoInts(_:_:)` function swaps the original value of `b` into `a`, and the original value of `a` into `b`. You can call this function to swap the values in two `Int` variables:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \
  (anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The `swapTwoInts(_:_:)` function is useful, but it can only be used with `Int` values. If you want to swap two `String` values, or two `Double` values, you have to write more functions, such as the `swapTwoStrings(_:_:)` and `swapTwoDoubles(_:_:)` functions shown below:

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

You may have noticed that the bodies of the `swapTwoInts(_:_:)`, `swapTwoStrings(_:_:)`, and `swapTwoDoubles(_:_:)` functions are identical. The only difference is the type of the values that they accept (`Int`, `String`, and `Double`).

It's more useful, and considerably more flexible, to write a single function that swaps two values of *any* type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

> NOTE

> In all three functions, the types of `a` and `b` must be the same. If `a` and `b` aren't of the same type, it isn't possible to swap their values. Swift is a type-safe language, and doesn't allow (for example) a variable of type `String` and a variable of type `Double` to swap values with each other. Attempting to do so results in a compile-time error.

Generic Functions

Generic functions can work with any type. Here's a generic version of the `swapTwoInts(_:_:)` function from above, called `swapTwoValues(_:_:)`:

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The body of the `swapTwoValues(_:_:)` function is identical to the body of the `swapTwoInts(_:_:)` function. However, the first line of `swapTwoValues(_:_:)` is slightly different from `swapTwoInts(_:_:)`. Here's how the first lines compare:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

The generic version of the function uses a *placeholder* type name (called `T`, in this case) instead of an *actual* type name (such as `Int`, `String`, or `Double`). The placeholder type name doesn't say anything about what `T` must be, but it *does* say that both `a` and `b` must be of the same type `T`, whatever `T` represents. The actual type to use in place of `T` is determined each time the `swapTwoValues(_:_:)` function is called.

The other difference between a generic function and a nongeneric function is that the generic function's name (`swapTwoValues(_:_:)`) is followed by the placeholder type name (`T`) inside angle brackets (`<T>`). The brackets tell Swift that `T` is a placeholder type name within the `swapTwoValues(_:_:)` function definition. Because `T` is a placeholder, Swift doesn't look for an actual type called `T`.

The `swapTwoValues(_:_:)` function can now be called in the same way as `swapTwoInts`, except that it can be passed two values of *any* type, as long as both of those values are of the same type as each other. Each time `swapTwoValues(_:_:)` is called, the type to use for `T` is inferred from the types of values passed to the function.

In the two examples below, `T` is inferred to be `Int` and `String` respectively:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now
  "hello"
```

> NOTE
>
> The `swapTwoValues(_:_:)` function defined above is inspired by a generic function called `swap`, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the `swapTwoValues(_:_:)` function in your own code, you can use Swift's existing `swap(_:_:)` function rather than providing your own implementation.

Type Parameters

In the `swapTwoValues(_:_:)` example above, the placeholder type `T` is an example of a *type parameter*. Type parameters specify and name a placeholder type, and are written immediately after the function's name, between a pair of matching angle brackets (such as <T>).

Once you specify a type parameter, you can use it to define the type of a function's parameters (such as the `a` and `b` parameters of the `swapTwoValues(_:_:)` function), or as the function's return type, or as a type annotation within the body of the function. In each case, the type parameter is replaced with an *actual* type whenever the function is called. (In the `swapTwoValues(_:_:)` example above, `T` was replaced with `Int` the first time the function was called, and was replaced with `String` the second time it was called.)

You can provide more than one type parameter by writing multiple type parameter names within the angle brackets, separated by commas.

Naming Type Parameters

In most cases, type parameters have descriptive names, such as `Key` and `Value` in `Dictionary<Key, Value>` and `Element` in `Array<Element>`, which tells the reader about the relationship between the type parameter and the generic type or function it's used in. However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as `T`, `U`, and `V`, such as `T` in the `swapTwoValues(_:_:)` function above.

> NOTE
>
> Always give type parameters upper camel case names (such as `T` and `MyTypeParameter`) to indicate that they're a placeholder for a *type*, not a value.

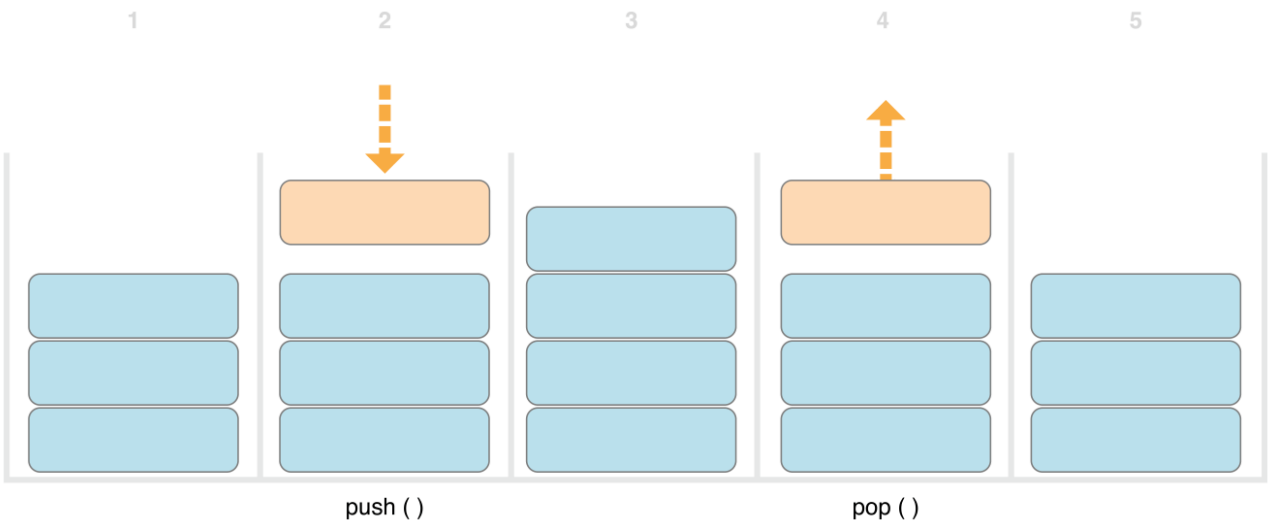
Generic Types

In addition to generic functions, Swift enables you to define your own *generic types*. These are custom classes, structures, and enumerations that can work with *any* type, in a similar way to `Array` and `Dictionary`.

This section shows you how to write a generic collection type called `Stack`. A stack is an ordered set of values, similar to an array, but with a more restricted set of operations than Swift's `Array` type. An array allows new items to be inserted and removed at any location in the array. A stack, however, allows new items to be appended only to the end of the collection (known as *pushing* a new value on to the stack). Similarly, a stack allows items to be removed only from the end of the collection (known as *popping* a value off the stack).

> NOTE
>
> The concept of a stack is used by the `UINavigationController` class to model the view controllers in its navigation hierarchy. You call the `UINavigationController` class `pushViewController(_:animated:)` method to add (or push) a view controller on to the navigation stack, and its `popViewControllerAnimated(_:)` method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict "last in, first out" approach to managing a collection.

The illustration below shows the push and pop behavior for a stack:



1. There are currently three values on the stack.
2. A fourth value is pushed onto the top of the stack.
3. The stack now holds four values, with the most recent one at the top.
4. The top item in the stack is popped.
5. After popping a value, the stack once again holds three values.

Here's how to write a nongeneric version of a stack, in this case for a stack of `Int` values:

```
1 struct IntStack {
```

```
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

This structure uses an `Array` property called `items` to store the values in the stack. `Stack` provides two methods, `push` and `pop`, to push and pop values on and off the stack. These methods are marked as `mutating`, because they need to modify (or *mutate*) the structure's `items` array.

The `IntStack` type shown above can only be used with `Int` values, however. It would be much more useful to define a *generic* `Stack` class, that can manage a stack of *any* type of value.

Here's a generic version of the same code:

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Note how the generic version of `Stack` is essentially the same as the nongeneric version, but with a type parameter called `Element` instead of an actual type of `Int`. This type parameter is written within a pair of angle brackets (`<Element>`) immediately after the structure's name.

`Element` defines a placeholder name for a type to be provided later. This future type can be referred to as `Element` anywhere within the structure's definition. In this case, `Element` is used as a placeholder in three places:

- To create a property called `items`, which is initialized with an empty array of

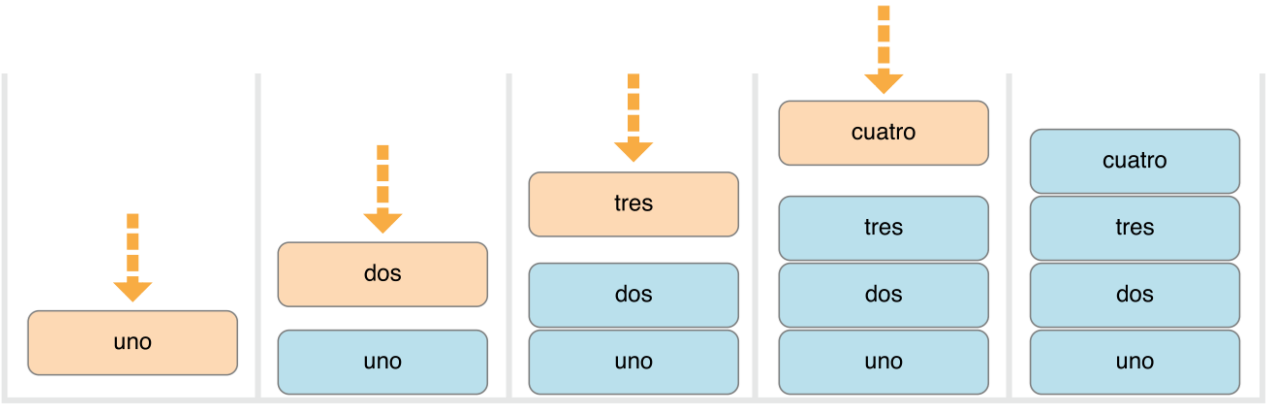
values of type `Element`
- To specify that the `push(_:)` method has a single parameter called `item`, which must be of type `Element`
- To specify that the value returned by the `pop()` method will be a value of type `Element`

Because it's a generic type, `Stack` can be used to create a stack of *any* valid type in Swift, in a similar manner to `Array` and `Dictionary`.

You create a new `Stack` instance by writing the type to be stored in the stack within angle brackets. For example, to create a new stack of strings, you write `Stack<String>()`:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

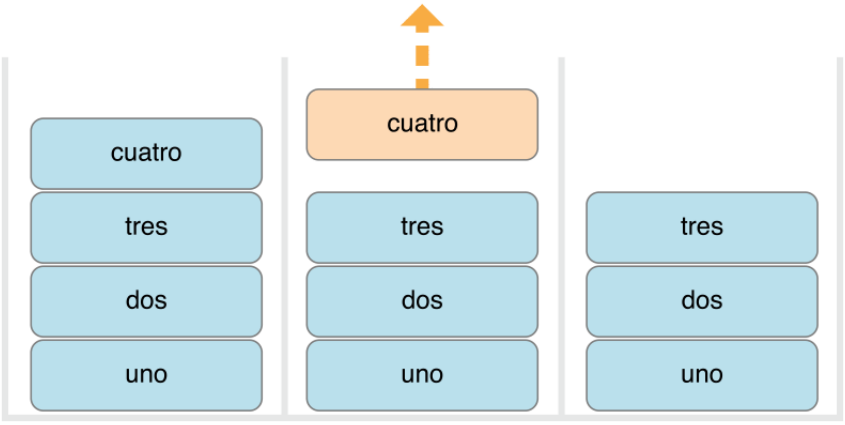
Here's how `stackOfStrings` looks after pushing these four values on to the stack:



Popping a value from the stack removes and returns the top value, `"cuatro"`:

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now
  contains 3 strings
```

Here's how the stack looks after popping its top value:



Extending a Generic Type

When you extend a generic type, you don't provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the *original* type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

The following example extends the generic `Stack` type to add a read-only computed property called `topItem`, which returns the top item on the stack without popping it from the stack:

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

The `topItem` property returns an optional value of type `Element`. If the stack is empty, `topItem` returns `nil`; if the stack isn't empty, `topItem` returns the final item in the `items` array.

Note that this extension doesn't define a type parameter list. Instead, the `Stack` type's existing type parameter name, `Element`, is used within the extension to indicate the optional type of the `topItem` computed property.

The `topItem` computed property can now be used with any `Stack` instance to access and query its top item without removing it.

```swift
if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}
// Prints "The top item on the stack is tres."
```

Extensions of a generic type can also include requirements that instances of the extended type must satisfy in order to gain the new functionality, as discussed in Extensions with a Generic Where Clause below.

Type Constraints

The `swapTwoValues(_:_:)` function and the `Stack` type can work with any type. However, it's sometimes useful to enforce certain *type constraints* on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift's `Dictionary` type places a limitation on the types that can be used as keys for a dictionary. As described in Dictionaries, the type of a dictionary's keys must be *hashable*. That is, it must provide a way to make itself uniquely representable. `Dictionary` needs its keys to be hashable so that it can check whether it already contains a value for a particular key. Without this requirement, `Dictionary` could not tell whether it should insert or replace a value for a particular key, nor would it be able to find a value for a given key that is already in the dictionary.

This requirement is enforced by a type constraint on the key type for `Dictionary`, which specifies that the key type must conform to the `Hashable` protocol, a special protocol defined in the Swift standard library. All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default.

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming. Abstract concepts like `Hashable` characterize types in terms of their conceptual characteristics, rather than their concrete type.

Type Constraint Syntax

You write type constraints by placing a single class or protocol constraint after a type parameter's name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below (although the syntax is the same for generic types):

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T,
  someU: U) {
    // function body goes here
}
```

The hypothetical function above has two type parameters. The first type parameter, `T`, has a type constraint that requires `T` to be a subclass of `SomeClass`. The second type parameter, `U`, has a type constraint that requires `U` to conform to the protocol `SomeProtocol`.

Type Constraints in Action

Here's a nongeneric function called `findIndex(ofString:in:)`, which is given a `String` value to find and an array of `String` values within which to find it. The `findIndex(ofString:in:)` function returns an optional `Int` value, which will be the index of the first matching string in the array if it's found, or `nil` if the string can't be found:

```
func findIndex(ofString valueToFind: String, in array:
  [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The `findIndex(ofString:in:)` function can be used to find a string value in an array of strings:

```
let strings = ["cat", "dog", "llama", "parakeet",
  "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in:
```

```
  strings) {
    print("The index of llama is \(foundIndex)")
}
// Prints "The index of llama is 2"
```

The principle of finding the index of a value in an array isn't useful only for strings, however. You can write the same functionality as a generic function by replacing any mention of strings with values of some type `T` instead.

Here's how you might expect a generic version of `findIndex(ofString:in:)`, called `findIndex(of:in:)`, to be written. Note that the return type of this function is still `Int?`, because the function returns an optional index number, not an optional value from the array. Be warned, though—this function doesn't compile, for reasons explained after the example:

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

This function doesn't compile as written above. The problem lies with the equality check, "`if value == valueToFind`". Not every type in Swift can be compared with the equal to operator (`==`). If you create your own class or structure to represent a complex data model, for example, then the meaning of "equal to" for that class or structure isn't something that Swift can guess for you. Because of this, it isn't possible to guarantee that this code will work for *every* possible type `T`, and an appropriate error is reported when you try to compile the code.

All is not lost, however. The Swift standard library defines a protocol called `Equatable`, which requires any conforming type to implement the equal to operator (`==`) and the not equal to operator (`!=`) to compare any two values of that type. All of Swift's standard types automatically support the `Equatable` protocol.

Any type that is `Equatable` can be used safely with the `findIndex(of:in:)` function, because it's guaranteed to support the equal to operator. To express this fact, you write a type constraint of `Equatable` as part of the type parameter's definition when you define the function:

```
func findIndex<T: Equatable>(of valueToFind: T, in array:
  [T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The single type parameter for `findIndex(of:in:)` is written as `T: Equatable`, which means "any type `T` that conforms to the `Equatable` protocol."

The `findIndex(of:in:)` function now compiles successfully and can be used with any type that is `Equatable`, such as `Double` or `String`:

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1,
  0.25])
// doubleIndex is an optional Int with no value, because 9.3
  isn't in the array
let stringIndex = findIndex(of: "Andrea", in: ["Mike",
  "Malcolm", "Andrea"])
// stringIndex is an optional Int containing a value of 2
```

Associated Types

When defining a protocol, it's sometimes useful to declare one or more associated types as part of the protocol's definition. An *associated type* gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type isn't specified until the protocol is adopted. Associated types are specified with the `associatedtype` keyword.

Associated Types in Action

Here's an example of a protocol called `Container`, which declares an associated type called `Item`:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

The `Container` protocol defines three required capabilities that any container must provide:

- It must be possible to add a new item to the container with an `append(_:)` method.
- It must be possible to access a count of the items in the container through a `count` property that returns an `Int` value.
- It must be possible to retrieve each item in the container with a subscript that takes an `Int` index value.

This protocol doesn't specify how the items in the container should be stored or what type they're allowed to be. The protocol only specifies the three bits of functionality that any type must provide in order to be considered a `Container`. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the `Container` protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

To define these requirements, the `Container` protocol needs a way to refer to the type of the elements that a container will hold, without knowing what that type is for a specific container. The `Container` protocol needs to specify that any value passed to the `append(_:)` method must have the same type as the container's element type, and that the value returned by the container's subscript will be of the same type as the container's element type.

To achieve this, the `Container` protocol declares an associated type called `Item`, written as `associatedtype Item`. The protocol doesn't define what `Item` is—that information is left for any conforming type to provide. Nonetheless, the `Item` alias provides a way to refer to the type of the items in a `Container`, and to define a type for use with the `append(_:)` method and subscript, to ensure that the expected behavior of any `Container` is enforced.

Here's a version of the nongeneric `IntStack` type from Generic Types above, adapted to conform to the `Container` protocol:

```
struct IntStack: Container {
    // original IntStack implementation
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

The `IntStack` type implements all three of the `Container` protocol's requirements, and in each case wraps part of the `IntStack` type's existing functionality to satisfy these requirements.

Moreover, `IntStack` specifies that for this implementation of `Container`, the appropriate `Item` to use is a type of `Int`. The definition of `typealias Item = Int` turns the abstract type of `Item` into a concrete type of `Int` for this implementation of the `Container` protocol.

Thanks to Swift's type inference, you don't actually need to declare a concrete `Item` of `Int` as part of the definition of `IntStack`. Because `IntStack` conforms to all of the requirements of the `Container` protocol, Swift can infer the appropriate `Item` to use, simply by looking at the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Indeed, if you delete the `typealias Item = Int` line from the code above, everything still works, because it's clear what type should be used for `Item`.

You can also make the generic `Stack` type conform to the `Container` protocol:

```
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

This time, the type parameter `Element` is used as the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Swift can therefore infer that `Element` is the appropriate type to use as the `Item` for this particular container.

Extending an Existing Type to Specify an Associated Type

You can extend an existing type to add conformance to a protocol, as described in Adding Protocol Conformance with an Extension. This includes a protocol with an associated type.

Swift's `Array` type already provides an `append(_:)` method, a `count` property, and a subscript with an `Int` index to retrieve its elements. These three capabilities match the requirements of the `Container` protocol. This means that you can extend `Array` to conform to the `Container` protocol simply by declaring that `Array` adopts the protocol. You do this with an empty extension, as described in Declaring Protocol Adoption with an Extension:

```
extension Array: Container {}
```

Array's existing `append(_:)` method and subscript enable Swift to infer the appropriate type to use for `Item`, just as for the generic `Stack` type above. After defining this extension, you can use any `Array` as a `Container`.

Adding Constraints to an Associated Type

You can add type constraints to an associated type in a protocol to require that conforming types satisfy those constraints. For example, the following code defines a version of `Container` that requires the items in the container to be equatable.

```
protocol Container {
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

To conform to this version of `Container`, the container's `Item` type has to conform to the `Equatable` protocol.

Using a Protocol in Its Associated Type's Constraints

A protocol can appear as part of its own requirements. For example, here's a protocol that refines the `Container` protocol, adding the requirement of a `suffix(_:)` method. The `suffix(_:)` method returns a given number of elements from the end of the container, storing them in an instance of the `Suffix` type.

```
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where
  Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

In this protocol, `Suffix` is an associated type, like the `Item` type in the `Container` example above. `Suffix` has two constraints: It must conform to the `SuffixableContainer` protocol (the protocol currently being defined), and its `Item` type must be the same as the container's `Item` type. The constraint on `Item` is a generic `where` clause, which is discussed in Associated Types with a Generic Where Clause below.

Here's an extension of the `Stack` type from Generic Types above that adds conformance to the `SuffixableContainer` protocol:

```
extension Stack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack {
        var result = Stack()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack.
}
var stackOfInts = Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
let suffix = stackOfInts.suffix(2)
// suffix contains 20 and 30
```

In the example above, the `Suffix` associated type for `Stack` is also `Stack`, so the suffix operation on `Stack` returns another `Stack`. Alternatively, a type that conforms to `SuffixableContainer` can have a `Suffix` type that's different from itself—meaning the suffix operation can return a different type. For example, here's an extension to the nongeneric `IntStack` type that adds `SuffixableContainer` conformance, using `Stack<Int>` as its suffix type instead of `IntStack`:

```
extension IntStack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack<Int> {
        var result = Stack<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack<Int>.
}
```

Generic Where Clauses

Type constraints, as described in Type Constraints, enable you to define requirements on the type parameters associated with a generic function, subscript, or type.

It can also be useful to define requirements for associated types. You do this by defining a *generic where clause*. A generic `where` clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same. A generic `where` clause starts with the `where` keyword, followed by constraints for associated types or equality relationships between types and associated types. You write a generic `where` clause right before the opening curly brace of a type or function's body.

The example below defines a generic function called `allItemsMatch`, which checks to see if two `Container` instances contain the same items in the same order. The function returns a Boolean value of `true` if all items match and a value of `false` if they don't.

The two containers to be checked don't have to be the same type of container (although they can be), but they do have to hold the same type of items. This requirement is expressed through a combination of type constraints and a generic `where` clause:

```
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

        // Check that both containers contain the same
  number of items.
        if someContainer.count != anotherContainer.count {
            return false
        }

        // Check each pair of items to see if they're
  equivalent.
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // All items match, so return true.
        return true
}
```

This function takes two arguments called `someContainer` and `anotherContainer`. The `someContainer` argument is of type `C1`, and the `anotherContainer` argument is of type `C2`. Both `C1` and `C2` are type parameters for two container types to be determined when the function is called.

The following requirements are placed on the function's two type parameters:

- `C1` must conform to the `Container` protocol (written as `C1: Container`).
- `C2` must also conform to the `Container` protocol (written as `C2: Container`).
- The `Item` for `C1` must be the same as the `Item` for `C2` (written as `C1.Item == C2.Item`).
- The `Item` for `C1` must conform to the `Equatable` protocol (written as `C1.Item: Equatable`).

The first and second requirements are defined in the function's type parameter list, and the third and fourth requirements are defined in the function's generic `where` clause.

These requirements mean:

- `someContainer` is a container of type `C1`.
- `anotherContainer` is a container of type `C2`.
- `someContainer` and `anotherContainer` contain the same type of items.
- The items in `someContainer` can be checked with the not equal operator (`!=`) to see if they're different from each other.

The third and fourth requirements combine to mean that the items in `anotherContainer` can *also* be checked with the `!=` operator, because they're exactly the same type as the items in `someContainer`.

These requirements enable the `allItemsMatch(_:_:)` function to compare the two containers, even if they're of a different container type.

The `allItemsMatch(_:_:)` function starts by checking that both containers contain the same number of items. If they contain a different number of items, there's no way that they can match, and the function returns `false`.

After making this check, the function iterates over all of the items in `someContainer` with a `for-in` loop and the half-open range operator (`..<`). For each item, the function checks whether the item from `someContainer` isn't equal to the corresponding item in `anotherContainer`. If the two items aren't equal, then the two containers don't match, and the function returns `false`.

If the loop finishes without finding a mismatch, the two containers match, and the function returns `true`.

Here's how the `allItemsMatch(_:_:)` function looks in action:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

```

```swift
if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
// Prints "All items match."
```

The example above creates a `Stack` instance to store `String` values, and pushes three strings onto the stack. The example also creates an `Array` instance initialized with an array literal containing the same three strings as the stack. Even though the stack and the array are of a different type, they both conform to the `Container` protocol, and both contain the same type of values. You can therefore call the `allItemsMatch(_:_:)` function with these two containers as its arguments. In the example above, the `allItemsMatch(_:_:)` function correctly reports that all of the items in the two containers match.

Extensions with a Generic Where Clause

You can also use a generic `where` clause as part of an extension. The example below extends the generic `Stack` structure from the previous examples to add an `isTop(_:)` method.

```swift
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```

This new `isTop(_:)` method first checks that the stack isn't empty, and then compares the given item against the stack's topmost item. If you tried to do this without a generic `where` clause, you would have a problem: The implementation of `isTop(_:)` uses the `==` operator, but the definition of `Stack` doesn't require its items to be equatable, so using the `==` operator results in a compile-time error. Using a generic `where` clause lets you add a new requirement to the extension, so that the extension adds the `isTop(_:)` method only when the items in the stack are equatable.

Here's how the `isTop(_:)` method looks in action:

```
if stackOfStrings.isTop("tres") {
    print("Top element is tres.")
} else {
    print("Top element is something else.")
}
// Prints "Top element is tres."
```

If you try to call the `isTop(_:)` method on a stack whose elements aren't equatable, you'll get a compile-time error.

```
struct NotEquatable { }
var notEquatableStack = Stack<NotEquatable>()
let notEquatableValue = NotEquatable()
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue)  // Error
```

You can use a generic `where` clause with extensions to a protocol. The example below extends the `Container` protocol from the previous examples to add a `startsWith(_:)` method.

```
extension Container where Item: Equatable {
    func startsWith(_ item: Item) -> Bool {
        return count >= 1 && self[0] == item
    }
}
```

The `startsWith(_:)` method first makes sure that the container has at least one item, and then it checks whether the first item in the container matches the given item. This new `startsWith(_:)` method can be used with any type that conforms to the `Container` protocol, including the stacks and arrays used above, as long as the container's items are equatable.

```
if [9, 9, 9].startsWith(42) {
    print("Starts with 42.")
} else {
    print("Starts with something else.")
}
// Prints "Starts with something else."
```

The generic `where` clause in the example above requires `Item` to conform to a protocol, but you can also write a generic `where` clauses that require `Item` to be a specific type. For example:

```
extension Container where Item == Double {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
print([1260.0, 1200.0, 98.6, 37.0].average())
// Prints "648.9"
```

This example adds an `average()` method to containers whose `Item` type is `Double`. It iterates over the items in the container to add them up, and divides by the container's count to compute the average. It explicitly converts the count from `Int` to `Double` to be able to do floating-point division.

You can include multiple requirements in a generic `where` clause that is part of an extension, just like you can for a generic `where` clause that you write elsewhere. Separate each requirement in the list with a comma.

Associated Types with a Generic Where Clause

You can include a generic `where` clause on an associated type. For example, suppose you want to make a version of `Container` that includes an iterator, like what the `Sequence` protocol uses in the standard library. Here's how you write that:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }

    associatedtype Iterator: IteratorProtocol where
  Iterator.Element == Item
    func makeIterator() -> Iterator
}
```

The generic `where` clause on `Iterator` requires that the iterator must traverse over elements of the same item type as the container's items, regardless of the iterator's type. The `makeIterator()` function provides access to a container's iterator.

For a protocol that inherits from another protocol, you add a constraint to an inherited associated type by including the generic `where` clause in the protocol declaration. For example, the following code declares a `ComparableContainer` protocol that requires `Item` to conform to `Comparable`:

```
protocol ComparableContainer: Container where Item:
  Comparable { }
```

Generic Subscripts

Subscripts can be generic, and they can include generic `where` clauses. You write the placeholder type name inside angle brackets after `subscript`, and you write a generic `where` clause right before the opening curly brace of the subscript's body. For example:

```
extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
```

This extension to the `Container` protocol adds a subscript that takes a sequence of indices and returns an array containing the items at each given index. This generic subscript is constrained as follows:

- The generic parameter `Indices` in angle brackets has to be a type that conforms to the `Sequence` protocol from the standard library.
- The subscript takes a single parameter, `indices`, which is an instance of that `Indices` type.
- The generic `where` clause requires that the iterator for the sequence must traverse over elements of type `Int`. This ensures that the indices in the sequence are the same type as the indices used for a container.

Taken together, these constraints mean that the value passed for the `indices` parameter is a sequence of integers.

‹ Protocols Opaque Types ›