Swift Standard Library Reference

Developer

Contents

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Types

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

String

A String represents an ordered collection of characters.

For a full discussion of String, see Strings and Characters.

Creating a String

init()

Constructs an empty string.

Declaration

```
init()
```

Discussion

Creating a string using this constructor:

```
let emptyString = String()
```

is equivalent to using double-quote convenience syntax:

```
let equivalentString = ""
```

init(count:, repeatedValue:)

Constructs a string with a single character repeated a given number of times.

Declaration

```
init(count sz: Int, repeatedValue c: Character)
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Discussion

The resulting string contains the supplied `repeatedValue` character, repeated `count` times:

```
let string = String(count: 5, repeatedValue: Character("a"))
// string is "aaaaa"
```

Querying a String

var isEmpty { get }

A Boolean value that determines whether the string is empty (read-only).

Declaration

```
var isEmpty: Bool { get }
```

Discussion

Use this read-only property to query whether the string is empty, which means it has no characters:

```
var string = "Hello, world!"
let firstCheck = string.isEmpty
// firstCheck is false

string = ""
let secondCheck = string.isEmpty
// secondCheck is true
```

hasPrefix(_ :) -> Bool

Returns a Boolean value that indicates whether the first characters in the receiver are the same as the characters in a given string.

Declaration

```
func hasPrefix(prefix: String) -> Bool
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Discussion

Use this method to determine whether the characters in a given string match exactly against the characters at the beginning of the receiver:

```
let string = "Hello, world"
let firstCheck = string.hasPrefix("Hello")
// firstCheck is true

let secondCheck = string.hasPrefix("hello")
// secondCheck is false
```

hasSuffix(_ :) -> Bool

Returns a Boolean value that indicates whether the last characters in the receiver are the same as the characters in a given string.

Declaration

```
func hasSuffix(suffix: String) -> Bool
```

Discussion

Use this method to determine whether the characters in a given string match exactly against the characters at the end of the receiver:

```
let string = "Hello, world"
let firstCheck = string.hasSuffix("world")
// firstCheck is true

let secondCheck = string.hasSuffix("World")
// secondCheck is false
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Converting Strings

toInt() -> Int?

Returns an optional integer, containing the result of attempting to convert the characters in the string into an integer value.

Declaration

```
func toInt() -> Int?
```

Discussion

Use this method to convert a string of characters into an integer value. The method returns an optional—if the conversion succeeded, the value will be the resulting integer; if the conversion failed, the value will be `nil`:

```
let string = "42"
if let number = string.toInt() {
    println("Got the number: \(number)")
} else {
    println("Couldn't convert to a number")
}
// prints "Got the number: 42"
```

Operators

+

Concatenates two strings, or a string and a character, or two characters.

Declaration

```
func + (lhs: String, rhs: String) -> String
func + (lhs: String, rhs: Character) -> String
func + (lhs: Character, rhs: String) -> String
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
func + (lhs: Character, rhs: Character) -> String
```

Discussion

Use the + operator to concatenate two strings:

```
let combination = "Hello " + "world"
// combination is "Hello world"
```

If the value supplied on the left hand side of the operator is an empty string, the resultant value is the unmodified value on the right hand side.

You can use the + operator with two strings as shown in the combination example, or with a string and a character in either order:

```
let exclamationPoint: Character = "!"
let charCombo = combination + exclamationPoint
// charCombo is "Hello world!"

let extremeCombo = exclamationPoint + charCombo
// extremeCombo is "!Hello world!"
```

Alternatively, you can combine two characters to form a string:

```
let first: Character = "a"
let second: Character = "b"
let result = first + second
// result is a String with the value "ab"
```

+=

Appends a string or character to an existing string.

Declaration

```
@assignment func += (inout lhs: String, rhs: String)
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
@assignment func += (inout lhs: String, rhs: Character)
```

Discussion

Use the += operator to append a string or character at the end of an existing string:

```
var string = "Hello "
string += "world!"
// string is "Hello world!"
```

If the initial string is empty, the resultant value is the unmodified `rhs` value.

You can use the += operator to append either another string, or a character:

```
var character: Character = "?"
string += character
// string is "Hello world!?"
```

You can only use the += operator to append if you declared the original string or character using the `var` keyword (that is, as a variable and not a constant):

```
let string = "Hello "
string += "world!"
// Error: could not find an overload for '+=' that accepts the supplied arguments
```

==

Determines the equality of two strings.

Declaration

```
func == (lhs: String, rhs: String) -> Bool
```

Discussion

Evaluates to `true` if the two string values contain exactly the same characters in exactly the same order:

```
let string1 = "Hello world!"
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
let string2 = "Hello" + " " + "world" + "!"
let result = string1 == string2
// result is true
```

<

Performs a lexicographical comparison to determine whether one string evaluates as less than another.

Declaration

```
func < (lhs: String, rhs: String) -> Bool
```

Discussion

Evaluates to `true` if the `lhs` value is less than the `rhs` value, by performing a lexicographical comparison of the characters:

```
let string1 = "Number 3"
let string2 = "Number 2"

let result1 = string1 < string2
// result1 is false

let result2 = string2 < string1
// result2 is true
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Array<T>

An Array is a generic type that manages an ordered collection of items, all of which must be of the same underlying type (T).

For more information about `Array`, see Collection Types.

Creating an Array

init()

Constructs an empty array of type T.

Declaration

```
init()
```

Discussion

Creating an array using this constructor:

```
var emptyArray = Array<Int>()
```

is equivalent to using the convenience syntax:

```
var equivalentEmptyArray = [Int]()
```

init(count:, repeatedValue:)

Constructs an array with a given number of elements, each initialized to the same value.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Declaration

```
init(count: Int, repeatedValue: T)
```

Discussion

The resulting array will have `count` elements in it, each initialized to the same value provided as the value for `repeatedValue`.

For example:

```
let numericArray = Array(count: 3, repeatedValue: 42)
// numericArray is [42, 42, 42]

let stringArray = Array(count: 2, repeatedValue: "Hello")
// stringArray is ["Hello", "Hello"]
```

Accessing Array Elements

subscript(Int) -> T { get set }

Gets or sets existing elements in an array using square bracket subscripting.

Declaration

```
subscript(index: Int) -> T { get { }    nonmutating set { } }
```

Discussion

Use subscripting to access the individual elements in any array:

```
var subscriptableArray = ["zero", "one", "two", "three"]
let zero = subscriptableArray[0]
// zero is "zero"
let three = subscriptableArray[3]
// three is "three"
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

If you declare the array using the `var` keyword (that is, as a variable), you can also use subscripting to change the value of any existing element in the array:

```
subscriptableArray[0] = "nothing"
subscriptableArray[3] = "three items"
```

It is not possible to insert additional items into the array using subscripting:

```
subscriptableArray[4] = "new item"
// Fatal error: Array index out of range
```

Instead, use the append() (page 16) function, or the += (page 26) operator.

You cannot use subscripting to change the value of any existing element in an array that you declare using the `let` keyword (that is, as a constant):

```
let constantArray = ["zero", "one", "two", "three"]
constantArray[0] = "nothing"
// Error: cannot mutate a constant array
```

subscript(Range<Int>) -> Slice<T>

Gets or sets a subrange of existing elements in an array using square bracket subscripting with an integer range.

Declaration

```
subscript(subRange: Range<Int>) -> Slice<T> { get { }    set { } }
```

Discussion

Use range subscripting to access one or more existing elements in any array:

```
var subscriptableArray = ["zero", "one", "two", "three"]
let subRange = subscriptableArray[1...3]
// subRange = ["one", "two", "three"]
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

If you declare the array using the `var` keyword (that is, as a variable), you can also use subscripting to change the values of a range of existing elements:

```
subscriptableArray[1...2] = ["oneone", "twotwo"]
// subscriptableArray is now ["zero", "oneone", "twotwo", "three"]
```

You do not need to provide the same number of items as you are replacing:

```
subscriptableArray[1...2] = []
// subscriptableArray is now ["zero", "three"]
```

It is not possible to insert additional items into the array using subscripting:

```
subscriptableArray[4...5] = ["four", "five"]
// Fatal error: Array replace: subRange extends past the end
```

You cannot use subscripting to change any values in an array that you declare using the `let` keyword (that is, as a constant):

```
let constantArray = ["zero", "one", "two", "three"]
constantArray[1...2] = []
// Error: cannot mutate a constant array
```

Instead, use the append() (page 16) function, or the += (page 26) operator.

Adding and Removing Elements

append()

Adds a new item as the last element in an existing array.

Declaration

```
mutating func append(newElement: T)
```

Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Discussion

Use this method to add a new item to an existing array. The new element will be added as the last item in the collection:

```
var array = [0, 1]
array.append(2)
// array is [0, 1, 2]

array.append(3)
// array is [0, 1, 2, 3]
```

You can only append new values to an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

```
let constantArray = [0, 1]
constantArray.append(2)
// Error: immutable value of type '[Int]' only has mutating members named 'append'
```

insert(_ :, atIndex:)

Inserts an element into the collection at a given index.

Declaration

```
mutating func insert(newElement: T, atIndex: Int)
```

Discussion

Use this method to insert a new element anywhere within the range of existing items, or as the last item:

```
var array = [1, 2, 3]
array.insert(0, atIndex: 0)
// array is [0, 1, 2, 3]
```

The index must be less than or equal to the number of items in the collection. If you attempt to insert an item at a greater index, you'll trigger an assertion:

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
array.insert(6, atIndex: 6)
// Fatal error: Array replace: subRange extends past the end
```

You can only insert new values in an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

```
let constantArray = [1, 2, 3]
constantArray.insert(0, atIndex: 0)
// Error: immutable value of type '[Int]' only has mutating members named 'insert'
```

removeAtIndex() -> T

Removes the element at the given index and returns it.

Declaration

```
mutating func removeAtIndex(index: Int) -> T
```

Discussion

Use this method to remove an element at the given `index`. The return value of the method is the element that was removed:

```
var array = [0, 1, 2, 3]
let removed = array.removeAtIndex(0)
// array is [1, 2, 3]
// removed is 0
```

The index must be less than the number of items in the collection. If you attempt to remove an item at a greater index, you'll trigger an assertion:

```
array.removeAtIndex(5)
// Fatal error: Array index out of range
```

You can only remove an element from an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
let constantArray = [0, 1, 2]
constantArray.removeAtIndex(0)
// Error: immutable value of type '[Int]' only has mutating members named
'removeAtIndex'
```

removeLast() -> T

Removes the last element from the collection and returns it.

Declaration

```
mutating func removeLast() -> T
```

Discussion

Use this method to remove the last element in the receiver. The return value of the method is the element that was removed:

```
var array = [1, 2, 3]
let removed = array.removeLast()
// array is [1, 2]
// removed is 3
```

There must be at least one element in the array before you call this method—if you call this method on an empty array, you'll trigger an assertion:

```
var emptyArray = [Int]()
let tryToRemove = emptyArray.removeLast()
// Fatal error: can't removeLast from an empty Array
```

You can only remove the last item from an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

```
let constantArray = [1, 2]
constantArray.removeLast()
// Error: immutable value of type '[Int]' only has mutating members named
'removeLast'
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

removeAll(keepCapacity: = false)

Removes all the elements from the collection, and by default clears the underlying storage buffer.

Declaration

```
mutating func removeAll(keepCapacity: Bool = false)
```

Discussion

Use this method to remove all of the elements in the array:

```
var array = [0, 1, 2, 3]
array.removeAll()
let count = array.count
// count is 0
```

Unless you specify otherwise, the underlying backing storage will be cleared.

You can only remove all items from an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

```
let constantArray = [1, 2]
constantArray.removeLast()
// Error: immutable value of type '[Int]' only has mutating members named 'removeAll'
```

reserveCapacity()

Ensures that the underlying storage can hold the given total number of elements.

Declaration

```
mutating func reserveCapacity(minimumCapacity: Int)
```

Discussion

Ensure that the array has enough contiguous underlying backing storage to store the total number of elements specified for `minimumCapacity`.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Querying an Array

var count { get }

An integer value that represents the number of elements in the array (read-only).

Declaration

```
var count: Int { get }
```

Discussion

Use this read-only property to query the number of elements in the array:

```
var array = ["zero", "one", "two"]
let firstCount = array.count
// firstCount is 3

array += "three"
let secondCount = array.count
// secondCount is 4
```

var isEmpty { get }

A Boolean value that determines whether the array is empty (read-only).

Declaration

```
var isEmpty: Bool { get }
```

Discussion

Use this read-only property to query whether the array is empty:

```
var array = ["zero", "one", "two", "three"]
let firstCheck = array.isEmpty
// firstCheck is false
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
array.removeAll()
let secondCheck = array.isEmpty
// secondCheck is true
```

var capacity { get }

An integer value that represents how many total elements the array can store without reallocation (read-only).

Declaration

```
var capacity: Int { get }
```

Discussion

Use this read-only property to query how many total elements the array can store without triggering a reallocation of the backing storage.

Algorithms

sort(_ :)

Sorts the receiver in place using a given closure to determine the order of a provided pair of elements.

Declaration

```
mutating func sort(isOrderedBefore: (T, T) -> Bool)
```

Discussion

Use this method to sort elements in the receiver. The closure that you supply for `isOrderedBefore` should return a Boolean value to indicate whether one element should be before (`true`) or after (`false`) another element:

```
var array = [3, 2, 5, 1, 4]
array.sort { $0 < $1 }
// array is [1, 2, 3, 4, 5]
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
array.sort { $1 < $0 }
// array is [5, 4, 3, 2, 1]
```

You can only use sort an array in place if you declared the array using the `var` keyword (that is, as a variable):

```
let constantArray = [3, 2, 5, 1, 4]
constantArray.sort { $0 < $1 }
// Error: immutable value of type [Int] only has mutating members named 'sort'
```

sorted(_ :) -> Array<T>

Returns an array containing elements from the receiver sorted using a given closure.

Declaration

```
func sorted(isOrderedBefore: (T, T) -> Bool) -> Array<T>
```

Discussion

Use this method to return a new array containing sorted elements from the receiver. The closure that you supply for `isOrderedBefore` should return a Boolean value to indicate whether one element should be before (`true`) or after (`false`) another element:

```
let array = [3, 2, 5, 1, 4]
let sortedArray = array.sorted { $0 < $1 }
// sortedArray is [1, 2, 3, 4, 5]

let descendingArray = array.sorted { $1 < $0 }
// descendingArray is [5, 4, 3, 2, 1]
```

reverse() -> Array<T>

Returns an array containing the elements of the receiver in reverse order by index.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Declaration

```
func reverse() -> Array<T>
```

Discussion

Use this method to return an array containing the elements of the receiver in reverse order; that is, the last item will be the first, the penultimate will be the second, and so on:

```
let array = [1, 2, 3, 4, 5]
let reversedArray = array.reverse()
// reversedArray = [5, 4, 3, 2, 1]
```

filter(_ :) -> Array<T>

Returns an array containing the elements of the receiver for which a provided closure indicates a match.

Declaration

```
func filter(includeElement: (T) -> Bool) -> Array<T>
```

Discussion

Use this method to return a new array by filtering an existing array. The closure that you supply for `includeElement:` should return a Boolean value to indicate whether an element should be included (`true`) or excluded (`false`) from the final collection:

```
let array = [0, 1, 2, 3, 4, 5, 6, 7]
let filteredArray = array.filter { $0 % 2 == 0 }
// filteredArray is [0, 2, 4, 6]
```

map<U>(_ :) -> Array<U>

Returns an array of elements built from the results of applying a provided transforming closure for each element.

Declaration

```
func map<U>(transform: (T) -> U) -> Array<U>
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Discussion

Use this method to return a new array containing the results of applying a provided closure to transform each element in the receiver:

```
let array = [0, 1, 2, 3]
let multipliedArray = array.map { $0 * 2 }
// multipliedArray is [0, 2, 4, 6]

let describedArray = array.map { "Number: \($0)" }
// describedArray is [Number: 0, Number: 1, Number: 2, Number: 3]
```

reduce<U>(_:, combine: (U, T)->U) -> U

Returns a single value representing the result of applying a provided reduction closure for each element.

Declaration

```
func reduce<U>(initial: U, combine: (U, T) -> U) -> U
```

Discussion

Use this method to reduce a collection of elements down to a single value by recursively applying the provided closure:

```
let array = [1, 2, 3, 4, 5]
let addResult = array.reduce(0) { $0 + $1 }
// addResult is 15

let multiplyResult = array.reduce(1) { $0 * $1 }
// multiplyResult is 120
```

The two results build as follows:

1. The arguments to the first closure call are the initial value you supply, and the first element in the collection.

   In the `addResult` case, that means an `initialValue` of `0` and a first element of 1: `{ 0 + 1 }`.

   In the `multiplyResult` case, that means an `initialValue` of `1` and a first element of 1: `{1 * 1}`.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

2. Next, the closure is called with the previous result as the first argument, and the second element as the second argument.

   In the `addResult` case, that means a result of `1` and the next item `2`: { 1 + 2 }.

   In the `multiplyResult` case, that means a result of `1` and the next item `2`: { 1 * 2 }.

3. The closures continue to be called with the previous result and the next element as arguments:

   In the `addResult` case, that means `{3 + 3}`, `{6 + 4}`, `{10 + 5}`, with a final result of `15`.

   In the `multiplyResult` case, that means `{2 * 3}`, `{6 * 4}`, `{24 * 5}`, with a final result of `120`.

Operators

+=

Appends an element or sequence of elements to an existing array.

Declaration

```
@assignment func += <U>(inout lhs: Array<T>, rhs: U)
```

Discussion

The += operator offers an easy way to append a single element or a sequence of elements to the end of an existing array:

```
var array = [0, 1, 2]
array += 3
// array is [0, 1, 2, 3]

array += [4, 5, 6]
// array is [0, 1, 2, 3, 4, 5, 6]
```

The type of the element or elements must match the type of the existing elements in the array:

```
array += "hello"
// Error: could not find an overload for '+=' that accepts the supplied arguments
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

You can only append new values to an array if you declared the array using the `var` keyword (that is, as a variable and not a constant):

```
let constantArray = [0, 1, 2]
constantArray += 3
// Error: could not find an overload for '+=' that accepts the supplied arguments
```

Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Dictionary<KeyType, ValueType>

A Dictionary is a generic type that manages an unordered collection of key-value pairs. All of a dictionary's keys must be compatible with its key type (`KeyType`). Likewise, all of a dictionary's values must be compatible with its value type (`ValueType`).

For more information about `Dictionary`, see Collection Types.

Creating a Dictionary

init(minimumCapacity: = 2)

Constructs an empty dictionary with capacity for at least the specified number of key-value pairs.

Declaration

```
init(minimumCapacity: Int = 2)
```

Discussion

You can create a dictionary using this constructor without specifying a value for `minimumCapacity`, in which case its default value of `2` will be used:

```
var emptyDictionary = Dictionary<String, Int>()
// constructs an empty dictionary ready to contain String keys and integer values
```

If you do provide a `minimumCapacity` value, note that the actual capacity reserved by the dictionary may be larger than the value you provide.

Creating a dictionary using this constructor is equivalent to using the convenience syntax:

```
var equivalentEmptyDictionary = [String: Int]()
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Accessing and Changing Dictionary Elements

subscript(KeyType) -> ValueType? { get set }

Gets, sets, or deletes a key-value pair in a dictionary using square bracket subscripting.

Declaration

```
subscript(key: KeyType) -> ValueType? { get { } set { } }
```

Discussion

Use subscripting to access the individual elements in any dictionary. The value returned from a dictionary's subscript is of type `ValueType?`—an optional with an underlying type of the dictionary's `ValueType`:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
let value = dictionary["two"]
// value is an optional integer with an underlying value of 2
```

In this example, `value` is of type `Int?`, not `Int`. Use optional binding to query and unwrap a dictionary subscript's return value if it is non-`nil`:

```
if let unwrappedValue = dictionary["three"] {
    println("The integer value for \"three\" was: \(unwrappedValue)")
}
// prints "The integer value for "three" was: 3"
```

You can also use subscripting to change the value associated with an existing key in the dictionary, add a new value, or remove the value for a key by setting it to `nil`:

```
dictionary["three"] = 33
// dictionary is now ["one": 1, "two": 2, "three": 33]

dictionary["four"] = 4
// dictionary is now ["one": 1, "two": 2, "three": 33, "four": 4]

dictionary["three"] = nil
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
// dictionary is now ["one": 1, "two": 2, "four": 4]
```

Values in a dictionary can be changed, added, or removed with subscripting only if the dictionary is defined with the `var` keyword (that is, if the dictionary is mutable):

```
let constantDictionary = ["one": 1, "two": 2, "three": 3]
constantDictionary["four"] = 4
// Error: could not find an overload for 'subscript' that accepts the supplied
arguments
```

updateValue(_:, forKey:) -> ValueType?

Inserts or updates a value for a given key and returns the previous value for that key if one existed, or `nil` if a previous value did not exist.

Declaration

```
mutating func updateValue(value: ValueType, forKey: KeyType) -> ValueType?
```

Discussion

Use this method to insert or update a value for a given key, as an alternative to subscripting. This method returns a value of type `ValueType?`—an optional with an underlying type of the dictionary's `ValueType`:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
let previousValue = dictionary.updateValue(22, forKey: "two")
// previousValue is an optional integer with an underlying value of 2
```

In this example, `previousValue` is of type `Int?`, not `Int`. Use optional binding to query and unwrap the return value if it is non-`nil`:

```
if let unwrappedPreviousValue = dictionary.updateValue(33, forKey: "three") {
    println("Replaced the previous value: \(unwrappedPreviousValue)")
} else {
    println("Added a new value")
}
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
// prints "Replaced the previous value: 3"
```

Values in a dictionary can be updated using this method only if the dictionary is defined with the `var` keyword (that is, if the dictionary is mutable):

```
let constantDictionary = ["one": 1, "two": 2, "three": 3]
constantDictionary.updateValue(4, forKey: "four")
// Error: immutable value of type '[String: Int]' only has mutating members named
 'updateValue'
```

removeValueForKey(_:) -> ValueType?

Removes the key-value pair for the specified key and returns its value, or `nil` if a value for that key did not previously exist.

Declaration

```
mutating func removeValueForKey(key: KeyType) -> ValueType?
```

Discussion

Use this method to remove a value for a given key, as an alternative to assigning the value `nil` using subscripting. This method returns a value of type `ValueType?`—an optional with an underlying type of the dictionary's `ValueType`:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
let previousValue = dictionary.removeValueForKey("two")
// previousValue is an optional integer with an underlying value of 2
```

In this example, `previousValue` is of type `Int?`, not `Int`. Use optional binding to query and unwrap the return value if it is non-`nil`:

```
if let unwrappedPreviousValue = dictionary.removeValueForKey("three") {
    println("Removed the old value: \(unwrappedPreviousValue)")
} else {
    println("Didn't find a value for the given key to delete")
}
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
// prints "Removed the old value: 3"
```

Values in a dictionary can be removed using this method only if the dictionary is defined with the `var` keyword (that is, if the dictionary is mutable):

```
let constantDictionary = ["one": 1, "two": 2, "three": 3]
constantDictionary.removeValueForKey("four")
// Error: immutable value of type '[String, Int]' only has mutating members named
 'removeValueForKey'
```

removeAll(keepCapacity: = false)

Removes all key-value pairs from the dictionary, and by default clears up the underlying storage buffer.

Declaration

```
mutating func removeAll(keepCapacity: Bool = default)
```

Discussion

Use this method to remove all of the key-value pairs in the dictionary:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
dictionary.removeAll()
// dictionary is now an empty dictionary
```

Unless you specify otherwise, the underlying backing storage will be cleared.

Values in a dictionary can be removed using this method only if the dictionary is defined with the `var` keyword (that is, if the dictionary is mutable):

```
let constantDictionary = ["one": 1, "two": 2, "three": 3]
constantDictionary.removeAll()
// Error: immutable value of type '[String, Int]' only has mutating members named
 'removeAll'
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Querying a Dictionary

var count { get }

An integer value that represents the number of key-value pairs in the dictionary (read-only).

Declaration

```
var count: Int { get }
```

Discussion

Use this read-only property to query the number of elements in the dictionary:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
let elementCount = dictionary.count
// elementCount is 3
```

var keys { get }

Returns an unordered iterable collection of all of a dictionary's keys.

Declaration

```
var keys: MapCollectionView<Dictionary<KeyType, ValueType>, KeyType> { get }
```

Discussion

Use this read-only property to retrieve an iterable collection of a dictionary's keys:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
for key in dictionary.keys {
    println("Key: \(key)")
}
// prints "Key: one\nKey: two\nKey: three"
```

To use a dictionary's keys with an API that takes an Array instance, initialize a new array with the `keys` property:

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
let array = Array(dictionary.keys)
// array is ["one", "two", "three"]
```

var values { get }

Returns an unordered iterable collection of all of a dictionary's values.

Declaration

```
var values: MapCollectionView<Dictionary<KeyType, ValueType>, ValueType> { get }
```

Discussion

Use this read-only property to retrieve an iterable collection of a dictionary's values:

```
var dictionary = ["one": 1, "two": 2, "three": 3]
for value in dictionary.values {
    println("Value: \(value)")
}
// prints "Value: 1\nValue: 2\nValue: 3"
```

To use a dictionary's values with an API that takes an Array instance, initialize a new array with the `values` property:

```
let array = Array(dictionary.values)
// array is [1, 2, 3]
```

Operators

==

Determines the equality of two dictionaries.

Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Declaration

```
func == <KeyType : Equatable, ValueType : Equatable>(lhs: Dictionary<KeyType,
ValueType>, rhs: Dictionary<KeyType, ValueType> -> Bool
```

Discussion

Evaluates to `true` if the two dictionaries contain exactly the same keys and values:

```
let dictionary1 = ["one": 1, "two": 2]
var dictionary2 = ["one": 1]
dictionary2["two"] = 2
let result = dictionary1 == dictionary2
// result is true
```

!=

Determines the inequality of two dictionaries.

Declaration

```
func != <KeyType : Equatable, ValueType : Equatable>(lhs: Dictionary<KeyType,
ValueType>, rhs: Dictionary<KeyType, ValueType> -> Bool
```

Discussion

Evaluates to `true` if the two dictionaries do not contain exactly the same keys and values:

```
let dictionary1 = ["one": 1, "two": 2]
let dictionary2 = ["one": 1]
let result = dictionary1 != dictionary2
// result is true
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Numeric Types

The Swift standard library contains many standard numeric types, suitable for storing Boolean, integer, and floating-point values.

Boolean Types

Swift includes one Boolean type, `Bool`, which may be either `true` or `false`.

Integer Types

The primary integer type in Swift is `Int`, which is word-sized. This means that it holds 32 bits on 32-bit platforms, and 64 bits on 64-bit platforms.

For the majority of use cases, you should use the base `Int` type.

If you require a type with a specific size or signedness, for example to work with raw data, Swift also includes the following types:

Type	Minimum Value	Maximum Value
`Int8`	`-128`	`127`
`Int16`	`-32,768`	`32,767`
`Int32`	`-2,147,483,648`	`2,147,483,647`
`Int64`	`-9,223,372,036,854,775,808`	`9,223,372,036,854,775,807`
`UInt8`	`0`	`255`
`UInt16`	`0`	`65,535`
`UInt32`	`0`	`4,294,967,295`
`UInt64`	`0`	`18,446,744,073,709,551,615`

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Floating Point Types

The primary floating-point type in Swift is `Double`, which uses 64 bits. If you do not require 64-bit precision, Swift also includes a 32-bit `Float` type.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Protocols

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Equatable

The Equatable protocol makes it possible to determine whether two values of the same type are considered to be equal.

There is one required operator overload defined in the protocol: ==.

Determining Equality

==

Determines the equality of two values of the same type.

Declaration

```
func == (lhs: Self, rhs: Self) -> Bool
```

Discussion

To conform to the protocol, you must provide an operator declaration for == at global scope. You should return `true` if the provided values are equal, otherwise `false`.

It is up to you to determine what equality means:

```
struct MyStruct: Equatable {
    var name = "Untitled"
}
func == (lhs: MyStruct, rhs: MyStruct) -> Bool {
    return lhs.name == rhs.name
}

let value1 = MyStruct()
var value2 = MyStruct()
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
let firstCheck = value1 == value2
// firstCheck is true

value2.name = "A New Name"
let secondCheck = value1 == value2
// secondCheck is false
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Comparable

The Comparable protocol makes it possible to compare two values of the same type.

There is one required operator overload defined in the protocol (<), as well as one defined in the inherited Equatable (page 39) protocol (==). By adopting the `Comparable` protocol and adding an operator overload for <, you automatically gain the ability to use >, <=, and >=.

Comparing Values

<

Determines whether one value is less than another value of the same type.

Declaration

```
func < (lhs: Self, rhs: Self) -> Bool
```

Discussion

To conform to the protocol, you must provide an operator declaration for < at global scope. You should return `true` if the `lhs` value is less than the `rhs` value, otherwise `false`.

It is up to you to determine what "less than" means:

```
struct MyStruct: Comparable {
    var name = "Untitled"
}
func < (lhs: MyStruct, rhs: MyStruct) -> Bool {
    return lhs.name < rhs.name
}
// and == operator overload too (required - see Equatable)
```

```
let value1 = MyStruct()
var value2 = MyStruct()
let firstCheck = value1 < value2
// firstCheck is false

value2.name = "A New Name"
let secondCheck = value2 < value1
// secondCheck is true
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Printable

The Printable protocol allows you to customize the textual representation of any type ready for printing (for example, to Standard Out).

A type must adopt this protocol if you wish to supply a value of that type to, for example, the print<T>(_:) (page 46) and println<T>(_:) (page 46) functions.

Describing the Value

description { get }

A string containing a suitable textual representation of the receiver (read-only).

Declaration

```
var description: String { get }
```

Discussion

This property is required for any type that adopts the `Printable` protocol. Use it to determine the textual representation to print when, for example, calling the print<T>(_:) (page 46) and println<T>(_:) (page 46) functions:

```
struct MyType: Printable {
    var name = "Untitled"
    var description: String {
    return "MyType: \(name)"
    }
}

let value = MyType()
println("Created a \(value)")
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
// prints "Created a MyType: Untitled"
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Free Functions

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Printing

There are two primary functions for printing values to Standard Out in the Swift standard library: `print()` and `println()`. The `println()` function is overloaded to receive either a value to print, or no value, in which case it prints a newline character.

Both functions are global free functions, which means they may be called in their own right without a receiver:

```
print("Hello, world!\n")
println("Hello, world!")
```

Primary Functions

print<T>(_:)

Writes the textual representation of a provided value to Standard Out.

Declaration

```
func print<T>(object: T)
```

Discussion

The value you supply for `object` must conform to the Printable (page 43) or DebugPrintable (page $@) protocol:

```
print("Hello, world\n")
// prints "Hello, world" followed by a new line character
```

println<T>(_:)

Writes the textual representation of a provided value, followed by a newline character, to Standard Out.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Declaration

```
func println<T>(object: T)
```

Discussion

The value you supply for `object` must conform to the Printable (page 43) or DebugPrintable (page $@) protocol:

```
println("Hello, world")
// prints "Hello, world" followed by a new line character
```

println()

Writes a newline character to Standard Out.

Declaration

```
func println()
```

Discussion

Call this function without any values to print a newline character to Standard Out:

```
print("Hello, world")
println()
// prints "Hello, world" followed by a new line character
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Algorithms

The Swift standard library contains a variety of algorithms to aid with common tasks, including sorting, finding, and many more.

More information forthcoming.

Sorting

sort<T: Comparable>(inout array: T[])

Sorts in place the elements of an array, all of which must be comparable and equatable.

Declaration

```
func sort<T: Comparable>(inout array: T[])
```

Discussion

Use this method to sort a mutable array in place using the standard < (page 41) operator. All values in the array must be of types that conform to the Comparable (page 41) protocol, which inherits from the Equatable (page 39) protocol:

```
var array = [5, 1, 6, 4, 2, 3]
sort(&array)
// array is [1, 2, 3, 4, 5, 6]
```

You can only use this method with an array declared using the `var` keyword (that is, a variable):

```
let constantArray = [5, 1, 6, 4, 2, 3]
sort(&constantArray)
// Fatal Error: cannot mutate a constant array
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

sort<T>(inout array: T[], pred: (T, T) -> Bool) -> T[]

Sorts in place an array of elements using a given predicate closure.

Declaration

```
func sort<T>(inout array: [T], predicate: (T, T) -> Bool)
```

Discussion

Use this method to sort a mutable array of elements in place using a closure. The closure must return a Boolean value to indicate whether the two items are in ascending order (`true`) or descending order (`false`):

```
var array = [5, 1, 3, 4, 2, 6]
sort(&array) { $0 > $1 }
// array is [6, 5, 4, 3, 2, 1]
```

You can only use this method with an array declared using the `var` keyword (that is, a variable):

```
let constantArray = [5, 1, 6, 4, 2, 3]
sort(&constantArray) { $0 > $1 }
// Fatal Error: cannot mutate a constant array
```

sorted<T: Comparable>(array: T[]) -> T[]

Returns a sorted array of elements, all of which must be comparable and equatable.

Declaration

```
func sorted<T: Comparable>(var array: T[]) -> T[]
```

Discussion

Use this method to sort using the standard < (page 41) operator. All values in the provided array must be of types that conform to the Comparable (page 41) protocol, which inherits from the Equatable (page 39) protocol:

```
let array = [5, 1, 6, 4, 2, 3]
let result = sorted(array)
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
// result is [1, 2, 3, 4, 5, 6]
```

sorted<T>(array: T[], pred: (T, T) -> Bool) -> T[]

Returns a sorted array of elements using a given predicate closure.

Declaration

```
func sorted<T>(var array: T[], pred: (T, T) -> Bool) -> T[]
```

Discussion

Use this method to sort an array of elements using a closure. The closure must return a Boolean value to indicate whether the two items are in ascending order (`true`) or descending order (`false`):

```
let array = [5, 1, 3, 4, 2, 6]
let result = sorted(array) { $0 > $1 }
// result is [6, 5, 4, 3, 2, 1]
```

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Document Revision History

This table describes the changes to *Swift Standard Library Reference*.

Date	Notes
2014-07-18	New document that describes the key structures, classes, protocols, and free functions available in the Swift Standard Library.

2014-07-18 | Copyright © 2014 Apple Inc. All Rights Reserved. Apple Confidential Information.

Apple Inc.
Copyright © 2014 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple and the Apple logo are trademarks of Apple Inc., registered in the U.S. and other countries.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.