Swift Standard Library Reference

Developer

Contents

Types

String

A `String` represents an ordered collection of characters.

For a full discussion of String, see Strings and Characters in *The Swift Programming Language*.

Creating a String

init()

Constructs an empty string.

Declaration

```
init()
```

Discussion

Creating a string using this initializer:

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

is equivalent to using double-quote convenience syntax:

```
let equivalentString = ""
```

init(count:repeatedValue:)

Constructs a string with a single character or Unicode scalar value repeated a given number of times.

Declaration

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

```
init(count: Int, repeatedValue c: UnicodeScalar)
```

Discussion

The resulting string contains the supplied `repeatedValue` character or Unicode scalar value, repeated the specified number of times:

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

let letterB: UnicodeScalar = "\u{0062}"
let anotherString = String(count: 5, repeatedValue: letterB)
//anotherString is "bbbbb"
```

Querying a String

isEmpty

A Boolean value that indicates 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(_:)

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

Declaration

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

Discussion

Use this method to determine whether a string has a particular prefix:

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

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

This method perfoms a character-by-character canonical equivalence comparision between the Unicode extended grapheme clusters in each string.

hasSuffix(_:)

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

Declaration

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

Discussion

Use this method to determine whether a string has a particular suffix:

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

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

This method perfoms a character-by-character canonical equivalence comparision between the Unicode extended grapheme clusters in each string.

Converting Strings

toInt()

Returns an optional integer that contains the result of attempting to convert the characters in the string to an integer value.

Declaration

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

Discussion

Use this method to convert a string to an integer value. The method returns an optional integer value (`Int?`)—if the conversion succeeds, the value is the resulting integer; if the conversion fails, the value is `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

+

Returns a string that contains the result of concatenating two strings.

Declaration

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

+=

Appends a string to an existing string.

Declaration

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

Discussion

Use the += operator to append a string to 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 only use the += operator to append a string to an existing string if you declared the existing string using the `var` keyword (that is, as a variable and not a constant):

```
let string = "Hello "
string += "world!"
// Error: cannot mutate a constant string
```

==

Returns a Boolean value that indicates whether the contents of two strings are the same.

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

This method perfoms a character-by-character canonical equivalence comparision between the Unicode extended grapheme clusters in each string.

<

Returns a Boolean value that indicates whether one string lexicographically precedes 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, locale-insensitive 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
```

This operation is locale-insensitive. As a result, if you are comparing strings to present to the end-user, you should typically use one of the locale-sensitive comparison methods of the `NSString` class instead. For example, see the `compare:options:range:locale:`, `localizedCompare:`, and `localizedCaseInsensitiveCompare:` methods of the `NSString` class.

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 in *The Swift Programming Language*.

Creating an Array

init()

Constructs an empty array of type `T`.

Declaration

```
init()
```

Discussion

Creating an array using this initializer:

```
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.

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)

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

Declaration

```
subscript(index: Int) -> T
```

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"
```

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
```

If you want to add additional items to an array, use the `append(_:)` (page 16) method or the `+=` (page 26) operator instead.

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>)

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>
```

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"]
```

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
```

If you want to add additional items to an array, use the `append(_:)` (page 16) method or the `+=` (page 26) operator instead.

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
```

Adding and Removing Elements

append(_:)

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

Declaration

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

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: cannot mutate a constant array
```

insert(_:atIndex:)

Inserts an element into the collection at a given index.

Declaration

```
mutating func insert(newElement: T, atIndex i: 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:

```
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: cannot mutate a constant array
```

removeAtIndex(_:)

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):

```
let constantArray = [0, 1, 2]
constantArray.removeAtIndex(0)
// Error: cannot mutate a constant array
```

removeLast()

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: cannot mutate a constant array
```

removeAll(keepCapacity:)

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

Declaration

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

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.removeAll()
// Error: cannot mutate a constant array
```

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`.

Querying an Array

count

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
```

isEmpty

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
```

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

capacity

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 array 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 array. 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]
```

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

You can only 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: cannot mutate a constant array
```

sorted(_:)

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

Declaration

```
func sorted(isOrderedBefore: (T, T) -> Bool) -> [T]
```

Discussion

Use this method to return a new array containing sorted elements from the array. 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()

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

Declaration

```
func reverse() -> [T]
```

Discussion

Use this method to return a new array containing the elements of the array 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(_:)

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

Declaration

```
func filter(includeElement: (T) -> Bool) -> [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(_:)

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

Declaration

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

Discussion

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

```
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(_:combine:)

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}`.

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 the elements of a sequence or collection to an existing array.

Declaration

```
func +=<T, S: SequenceType where S.Generator.Element == T>(inout lhs: [T], rhs:
 S)

func +=<T, C: CollectionType where C.Generator.Element == T>(inout lhs: [T], rhs:
 C)
```

Discussion

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

```
var array = [0, 1, 2, 3]

array += [4, 5, 6]

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

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

```
array += ["hello"]
// Error: 'array' contains elements of type 'Int', not 'String'.
```

You can only use the += operator to append the elements of a sequence or collection 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: cannot mutate a constant array
```

Dictionary<Key: Hashable, Value>

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 (`Key`). Likewise, all of a dictionary's values must be compatible with its value type (`Value`).

For more information about `Dictionary`, see Collection Types in *The Swift Programming Language*.

Creating a Dictionary

init()

Constructs an empty dictionary.

Declaration

```
init()
```

Discussion

Creating a dictionary using this initializer:

```
var emptyDictionary = Dictionary<String: Int>()
```

is equivalent to using the convenience syntax:

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

init(minimumCapacity:)

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

Declaration

```
init(minimumCapacity: Int)
```

Discussion

You can create a dictionary using this initializer by specifying a value for `minimumCapacity`:

```
var emptyDictionary = Dictionary<String, Int>(minimumCapacity: 2)
// constructs an empty dictionary ready to contain at least two pairs of String
keys and Int values
```

Note that the actual capacity reserved by the dictionary will be the smallest power of 2 that is greater than or equal to the value specified for `minimumCapacity`.

Accessing and Changing Dictionary Elements

subscript(key: Key)

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

Declaration

```
subscript(key: Key) -> Value?
```

Discussion

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

```
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
// 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: cannot mutate a constant dictionary
```

updateValue(_:forKey:)

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: Value, forKey key: Key) -> Value?
```

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 `Value?`—an optional with an underlying type of the dictionary's `Value`:

```
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")
}
// 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: cannot mutate a constant dictionary
```

removeValueForKey(_:)

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: Key) -> Value?
```

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 `Value?`—an optional with an underlying type of the dictionary's `Value`:

```
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")
}
// 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: cannot mutate a constant dictionary
```

removeAll(keepCapacity:)

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: cannot mutate a constant dictionary
```

Querying a Dictionary

count

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
```

keys

Returns an unordered iterable collection of all of a dictionary's keys (read-only).

Declaration

```
var keys: LazyBidirectionalCollection<MapCollectionView<[Key : Value], Key>> { 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
// Key: three
// Key: two
```

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

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

values

Returns an unordered iterable collection of all of a dictionary's values (read-only).

Declaration

```
var values: LazyBidirectionalCollection<MapCollectionView<[Key : Value], Value>>
{ 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
// Value: 3
// Value: 2
```

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, 3, 2]
```

Operators

==

Returns a Boolean value that indicates whether the contents of two dictionaries are the same.

Declaration

```
func ==<Key: Equatable, Value: Equatable>(lhs: [Key: Value], rhs: [Key: Value])
-> 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
```

!=

Returns a Boolean value that indicates whether the contents of two dictionaries are not the same.

Declaration

```
func !=<Key: Equatable, Value: Equatable>(lhs: [Key: Value], rhs: [Key: Value])
-> 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
```

Numeric Types

The Swift standard library contains many standard numeric types, suitable for storing various integer and floating-point values. The Swift standard library also contains a single Boolean type to store Boolean values.

Integer Types

As shown in the table below, the Swift standard library provides types for signed and unsigned integers in 8, 16, 32, and 64 bit forms. The standard library also provides two types of native word-sized integers: `Int` for signed integers and `UInt` for unsigned integers. For example, `Int` holds 32 bits on 32-bit platforms and 64 bits on a 64-bit platform. Similarly, `UInt` holds 32 bits on 32-bit platforms and 64 bits on a 64-bit platform.

The default inferred type of an integer literal is `Int`:

```
let intValue = 42
// intValue is of type Int
```

You should use the word-sized `Int` type to store integer values, unless you require a type with a specific size or signedness.

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`

Floating Point Types

The Swift standard library provides three signed floating-point number types: `Float` for 32-bit floating-point numbers, `Double` for 64-bit floating point numbers, and `Float80` for extended-precision 80-bit floating-point numbers.

The default inferred type of a floating-point literal is `Double`:

```
let floatingPointValue = -273.15
// floatingPointValue is of type Double
```

Boolean Type

The Swift standard library provides one Boolean type, `Bool`. `Bool` is a value type whose instances are either `true` or `false`.

Protocols

Equatable

The `Equatable` protocol makes it possible to determine whether two values of the same type are considered to be equal using the == (page 40) and `!=` operators.

The Swift standard library provides an implementation for the `!=` operator. As a result, types that conform to the `Equatable` protocol are required to implement the == (page 40) operator only.

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 implementation 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 for two values of the conforming type:

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

let value1 = MyStruct()
```

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

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

Comparable

The `Comparable` protocol makes it possible to compare two values of the same type using the < (page 42), >, <=, and >= relational operators.

The Swift standard library provides an implementation for the >, <=, >=, and != operators. As a result, types that conform to the `Comparable` protocol are required to implement the < (page 42) and == (page 40) operators only. (The == (page 40) operator is a requirement because `Comparable` inherits from the `Equatable` (page 40) protocol.)

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 implementation 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 for two values of the conforming type:

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

```
// Implementation of == required by Equatable
func ==(lhs: MyStruct, rhs: MyStruct) -> Bool {
    return lhs.name == rhs.name
}

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
```

Printable

The Printable protocol allows you to customize the textual representation of any type ready for printing to an output stream (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(_:)` (page 47) and `println(_:)` (page 47) functions.

Describing the Value

description

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(_:)` (page 47) and `println(_:)` (page 47) functions:

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

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

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

Free Functions

Printing

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

Primary Functions

print(_:)

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 44) or `DebugPrintable` (page $@) protocol:

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

println(_:)

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

Declaration

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

Discussion

The value you supply for `object` must conform to the `Printable` (page 44) 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
```

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(_:)

Sorts an array of comparable elements in place.

Declaration

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

Discussion

Use this function to sort a mutable array in place using the < (page 42) operator required by conformance to the `Comparable` (page 42) protocol. All elements in the array must be of types that conform to the `Comparable` (page 42) protocol, and the < (page 42) operator must be defined as a strict weak ordering over the elements:

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

You can only use this function 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
```

The sorting algorithm is not stable and thus can change the relative ordering of elements that compare equals.

sort(_:_:)

Sorts an array of elements in place according to a specified closure.

Declaration

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

Discussion

Use this function 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 function 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
```

The `isOrderedBefore` closure must define a strict weak ordering over the elements in the array.

The sorting algorithm is not stable and thus can change the relative ordering of elements for which `isOrderedBefore` does not establish an order.

sorted(_:)

Returns an array that contains the sorted elements of a sequence.

Declaration

```
func sorted<C: SequenceType where C.Generator.Element: Comparable>(source: C) ->
 [C.Generator.Element]
```

Discussion

Use this function to sort a sequence using the < (page 42) operator required by conformance to the `Comparable` (page 42) protocol. All elements in `source` must be of types that conform to the `Comparable` (page 42) protocol, and the < (page 42) operator must be defined as a strict weak ordering over the elements:

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

The sorting algorithm is not stable and thus can change the relative ordering of elements that compare equals.

sorted(_:_:)

Returns an array that contains the sorted elements of a sequence according to a specified closure.

Declaration

```
func sorted<C: SequenceType>(source: C, isOrderedBefore: (C.Generator.Element,
C.Generator.Element) -> Bool) -> [C.Generator.Element]
```

Discussion

Use this function to sort a sequence 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]
```

The `isOrderedBefore` closure must define a strict weak ordering over the elements in the array.

The sorting algorithm is not stable and thus can change the relative ordering of elements for which `isOrderedBefore` does not establish an order.

Finding

find(_:_:)

Returns an optional index that contains the result of attempting to find the first index where a given value appears in a specified collection.

Declaration

```
func find<C: CollectionType where C.Generator.Element: Equatable>(domain: C,
value: C.Generator.Element) -> C.Index?
```

Discussion

Use this function to find the first index where the specified value appears in a collection. The function returns an optional index value (`C.Index?`)—if the specified value isn't found, the returned index value is `nil`. All elements in the collection must be of types that conform to the `Equatable` (page 40) protocol.

```
let array = [5, 1, 6, 4, 2, 3]
let valueToFind = 6
if let index = find(array4, valueToFind) {
    println("Found the value \(valueToFind) at index \(index).")
} else {
    println("Couldn't find the value \(valueToFind) in the array.")
}
// prints "Found the value 6 at index 2."
```

Document Revision History

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

Date	Notes
2014-09-17	New document that describes the key structures, classes, protocols, and free functions available in the Swift Standard Library.

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