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
```