Structure

Dictionary

A collection whose elements are key-value pairs.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
@frozen
struct Dictionary<Key, Value> where Key : Hashable
```

Overview

A dictionary is a type of hash table, providing fast access to the entries it contains. Each entry in the table is identified using its key, which is a hashable type such as a string or number. You use that key to retrieve the corresponding value, which can be any object. In other languages, similar data types are known as hashes or associated arrays.

Create a new dictionary by using a dictionary literal. A dictionary literal is a comma-separated list of key-value pairs, in which a colon separates each key from its associated value, surrounded by square brackets. You can assign a dictionary literal to a variable or constant or pass it to a function that expects a dictionary.

Here's how you would create a dictionary of HTTP response codes and their related messages:

```
var responseMessages = [200: "OK",
                        403: "Access forbidden",
                        404: "File not found",
                        500: "Internal server error"]
```

The `responseMessages` variable is inferred to have type `[Int: String]`. The `Key` type of the dictionary is `Int`, and the `Value` type of the dictionary is `String`.

To create a dictionary with no key-value pairs, use an empty dictionary literal (`[:]`).

```swift
var emptyDict: [String: String] = [:]
```

Any type that conforms to the `Hashable` protocol can be used as a dictionary's `Key` type, including all of Swift's basic types. You can use your own custom types as dictionary keys by making them conform to the `Hashable` protocol.

Getting and Setting Dictionary Values

The most common way to access values in a dictionary is to use a key as a subscript. Subscripting with a key takes the following form:

```swift
print(responseMessages[200])
// Prints "Optional("OK")"
```

Subscripting a dictionary with a key returns an optional value, because a dictionary might not hold a value for the key that you use in the subscript.

The next example uses key-based subscripting of the `responseMessages` dictionary with two keys that exist in the dictionary and one that does not.

```swift
let httpResponseCodes = [200, 403, 301]
for code in httpResponseCodes {
    if let message = responseMessages[code] {
        print("Response \(code): \(message)")
    } else {
        print("Unknown response \(code)")
    }
}
// Prints "Response 200: OK"
// Prints "Response 403: Access forbidden"
// Prints "Unknown response 301"
```

You can also update, modify, or remove keys and values from a dictionary using the key-based subscript. To add a new key-value pair, assign a value to a key that isn't yet a part of the dictionary.

```swift
responseMessages[301] = "Moved permanently"
print(responseMessages[301])
// Prints "Optional("Moved permanently")"
```

Update an existing value by assigning a new value to a key that already exists in the dictionary. If you assign `nil` to an existing key, the key and its associated value are removed. The following example updates the value for the `404` code to be simply "Not found" and removes the key-value pair for the `500` code entirely.

```
responseMessages[404] = "Not found"
responseMessages[500] = nil
print(responseMessages)
// Prints "[301: "Moved permanently", 200: "OK", 403: "Access forbidden", 404: "Not
```

In a mutable `Dictionary` instance, you can modify in place a value that you've accessed through a keyed subscript. The code sample below declares a dictionary called `interestingNumbers` with string keys and values that are integer arrays, then sorts each array in-place in descending order.

```
var interestingNumbers = ["primes": [2, 3, 5, 7, 11, 13, 17],
                          "triangular": [1, 3, 6, 10, 15, 21, 28],
                          "hexagonal": [1, 6, 15, 28, 45, 66, 91]]
for key in interestingNumbers.keys {
    interestingNumbers[key]?.sort(by: >)
}

print(interestingNumbers["primes"]!)
// Prints "[17, 13, 11, 7, 5, 3, 2]"
```

Iterating Over the Contents of a Dictionary

Every dictionary is an unordered collection of key-value pairs. You can iterate over a dictionary using a `for`-`in` loop, decomposing each key-value pair into the elements of a tuple.

```
let imagePaths = ["star": "/glyphs/star.png",
                  "portrait": "/images/content/portrait.jpg",
                  "spacer": "/images/shared/spacer.gif"]

for (name, path) in imagePaths {
    print("The path to '\(name)' is '\(path)'.")
}
```

```
// Prints "The path to 'star' is '/glyphs/star.png'."
// Prints "The path to 'portrait' is '/images/content/portrait.jpg'."
// Prints "The path to 'spacer' is '/images/shared/spacer.gif'."
```

The order of key-value pairs in a dictionary is stable between mutations but is otherwise unpredictable. If you need an ordered collection of key-value pairs and don't need the fast key lookup that `Dictionary` provides, see the `KeyValuePairs` type for an alternative.

You can search a dictionary's contents for a particular value using the `contains(where:)` or `firstIndex(where:)` methods supplied by default implementation. The following example checks to see if `imagePaths` contains any paths in the `"/glyphs"` directory:

```
let glyphIndex = imagePaths.firstIndex(where: { $0.value.hasPrefix("/glyphs") })
if let index = glyphIndex {
    print("The '\(imagePaths[index].key)' image is a glyph.")
} else {
    print("No glyphs found!")
}
// Prints "The 'star' image is a glyph."
```

Note that in this example, `imagePaths` is subscripted using a dictionary index. Unlike the key-based subscript, the index-based subscript returns the corresponding key-value pair as a non-optional tuple.

```
print(imagePaths[glyphIndex!])
// Prints "(key: "star", value: "/glyphs/star.png")"
```

A dictionary's indices stay valid across additions to the dictionary as long as the dictionary has enough capacity to store the added values without allocating more buffer. When a dictionary outgrows its buffer, existing indices may be invalidated without any notification.

When you know how many new values you're adding to a dictionary, use the `init(minimumCapacity:)` initializer to allocate the correct amount of buffer.

Bridging Between Dictionary and NSDictionary

You can bridge between `Dictionary` and `NSDictionary` using the `as` operator. For bridging to be possible, the `Key` and `Value` types of a dictionary must be classes, `@objc` protocols, or types that bridge to Foundation types.

Bridging from `Dictionary` to `NSDictionary` always takes O(1) time and space. When the dictionary's `Key` and `Value` types are neither classes nor `@objc` protocols, any required bridging of elements occurs at the first access of each element. For this reason, the first operation that uses the contents of the dictionary may take O(*n*).

Bridging from `NSDictionary` to `Dictionary` first calls the `copy(with:)` method (`- copyWithZone:` in Objective-C) on the dictionary to get an immutable copy and then performs additional Swift bookkeeping work that takes O(1) time. For instances of `NSDictionary` that are already immutable, `copy(with:)` usually returns the same dictionary in O(1) time; otherwise, the copying performance is unspecified. The instances of `NSDictionary` and `Dictionary` share buffer using the same copy-on-write optimization that is used when two instances of `Dictionary` share buffer.

Topics

Creating a Dictionary

In addition to using a dictionary literal, you can also create a dictionary using these initializers.

`init()`

Creates an empty dictionary.

`init(minimumCapacity: Int)`

Creates an empty dictionary with preallocated space for at least the specified number of elements.

`init<S>(uniqueKeysWithValues: S)`

Creates a new dictionary from the key-value pairs in the given sequence.

`init<S>(S, uniquingKeysWith: (Value, Value) throws -> Value) rethrows`

Creates a new dictionary from the key-value pairs in the given sequence, using a combining closure to determine the value for any duplicate keys.

`init<S>(grouping: S, by: (S.Element) throws -> Key) rethrows`

Creates a new dictionary whose keys are the groupings returned by the given closure and whose values are arrays of the elements that returned each key.

Inspecting a Dictionary

`var isEmpty: Bool`

A Boolean value that indicates whether the dictionary is empty.

`var count: Int`

The number of key-value pairs in the dictionary.

`var capacity: Int`

The total number of key-value pairs that the dictionary can contain without allocating new storage.

Accessing Keys and Values

`subscript(Key) -> Value?`

Accesses the value associated with the given key for reading and writing.

`subscript(Key, default _: @autoclosure () -> Value) -> Value`

Accesses the value with the given key, falling back to the given default value if the key isn’t found.

`func index(forKey: Key) -> Dictionary<Key, Value>.Index?`

Returns the index for the given key.

`subscript(Dictionary<Key, Value>.Index) -> Dictionary<Key, Value>.Element`

Accesses the key-value pair at the specified position.

`var keys: Dictionary<Key, Value>.Keys`

A collection containing just the keys of the dictionary.

`var values: Dictionary<Key, Value>.Values`

A collection containing just the values of the dictionary.

`var first: Self.Element?`

The first element of the collection.

`func randomElement() -> Self.Element?`

Returns a random element of the collection.

`func randomElement<T>(using: inout T) -> Self.Element?`

Returns a random element of the collection, using the given generator as a source for randomness.

Adding Keys and Values

`func updateValue(Value, forKey: Key) -> Value?`

Updates the value stored in the dictionary for the given key, or adds a new key-value pair if the key does not exist.

`func merge([Key : Value], uniquingKeysWith: (Value, Value) throws -> Value) rethrows`

Merges the given dictionary into this dictionary, using a combining closure to determine the value for any duplicate keys.

`func merge<S>(S, uniquingKeysWith: (Value, Value) throws -> Value) rethrows`

Merges the key-value pairs in the given sequence into the dictionary, using a combining closure to determine the value for any duplicate keys.

`func merging([Key : Value], uniquingKeysWith: (Value, Value) throws -> Value) rethrows -> [Key : Value]`

Creates a dictionary by merging the given dictionary into this dictionary, using a combining closure to determine the value for duplicate keys.

`func merging<S>(S, uniquingKeysWith: (Value, Value) throws -> Value) rethrows -> [Key : Value]`

Creates a dictionary by merging key-value pairs in a sequence into the dictionary, using a combining closure to determine the value for duplicate keys.

`func reserveCapacity(Int)`

Reserves enough space to store the specified number of key-value pairs.

Removing Keys and Values

`func filter((Dictionary<Key, Value>.Element) throws -> Bool) rethrows -> [Key : Value]`

Returns a new dictionary containing the key-value pairs of the dictionary that satisfy the given predicate.

`func removeValue(forKey: Key) -> Value?`

Removes the given key and its associated value from the dictionary.

`func remove(at: Dictionary<Key, Value>.Index) -> Dictionary<Key, Value>.Element`

Removes and returns the key-value pair at the specified index.

`func removeAll(keepingCapacity: Bool)`

Removes all key-value pairs from the dictionary.

Comparing Dictionaries

`static func == ([Key : Value], [Key : Value]) -> Bool`

Returns a Boolean value indicating whether two values are equal.

`static func != (Self, Self) -> Bool`

Returns a Boolean value indicating whether two values are not equal.

Iterating over Keys and Values

`func forEach((Self.Element) throws -> Void) rethrows`

Calls the given closure on each element in the sequence in the same order as a `for-in` loop.

`func enumerated() -> EnumeratedSequence<Self>`

Returns a sequence of pairs (*n*, *x*), where *n* represents a consecutive integer starting at zero and *x* represents an element of the sequence.

`var lazy: LazySequence<Self>`

A sequence containing the same elements as this sequence, but on which some operations, such as `map` and `filter`, are implemented lazily.

`func makeIterator() -> Dictionary<Key, Value>.Iterator`

Returns an iterator over the dictionary's key-value pairs.

`var underestimatedCount: Int`

A value less than or equal to the number of elements in the collection.

Finding Elements

`func contains(where: (Self.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether the sequence contains an element that satisfies the given predicate.

`func allSatisfy((Self.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether every element of a sequence satisfies a given predicate.

`func first(where: (Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the first element of the sequence that satisfies the given predicate.

`func firstIndex(where: (Self.Element) throws -> Bool) rethrows -> Self.Index?`

Returns the first index in which an element of the collection satisfies the given predicate.

`func min(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the minimum element in the sequence, using the given predicate as the comparison between elements.

`func max(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the maximum element in the sequence, using the given predicate as the comparison between elements.

Transforming a Dictionary

`func mapValues<T>((Value) throws -> T) rethrows -> Dictionary<Key, T>`

Returns a new dictionary containing the keys of this dictionary with the values transformed by the given closure.

`func reduce<Result>(Result, (Result, Self.Element) throws -> Result) rethrows -> Result`

Returns the result of combining the elements of the sequence using the given closure.

`func reduce<Result>(into: Result, (inout Result, Self.Element) throws -> ()) rethrows -> Result`

Returns the result of combining the elements of the sequence using the given closure.

`func compactMap<ElementOfResult>((Self.Element) throws -> ElementOfResult?) rethrows -> [ElementOfResult]`

Returns an array containing the non-`nil` results of calling the given transformation with each element of this sequence.

`func compactMapValues<T>((Value) throws -> T?) rethrows -> Dictionary<Key, T>`

Returns a new dictionary containing only the key-value pairs that have non-`nil` values as the result of transformation by the given closure.

`func flatMap<SegmentOfResult>((Self.Element) throws -> SegmentOfResult) rethrows -> [SegmentOfResult.Element]`

Returns an array containing the concatenated results of calling the given transformation with each element of this sequence.

```
func flatMap<ElementOfResult>((Self.Element) throws -> ElementOfResult
?) rethrows -> [ElementOfResult]
```

```
func sorted(by: (Self.Element, Self.Element) throws -> Bool) rethrows -
> [Self.Element]
```

Returns the elements of the sequence, sorted using the given predicate as the comparison between elements.

```
func shuffled() -> [Self.Element]
```

Returns the elements of the sequence, shuffled.

```
func shuffled<T>(using: inout T) -> [Self.Element]
```

Returns the elements of the sequence, shuffled using the given generator as a source for randomness.

Performing Collection Operations

Order Dependent Operations on Dictionary

Perform order-dependent operations common to all collections, as implemented for `Dictionary`.

Encoding and Decoding

```
func encode(to: any Encoder) throws
```

Encodes the contents of this dictionary into the given encoder.

```
init(from: any Decoder) throws
```

Creates a new dictionary by decoding from the given decoder.

Describing a Dictionary

```
var description: String
```

A string that represents the contents of the dictionary.

```
var debugDescription: String
```

A string that represents the contents of the dictionary, suitable for debugging.

```
var customMirror: Mirror
```

A mirror that reflects the dictionary.

`func hash(into: inout Hasher)`

Hashes the essential components of this value by feeding them into the given hasher.

Using a Dictionary as a Data Value

`init?(from: MLDataValue.DictionaryType)`

Reference Types

Use bridged reference types when you need reference semantics or Foundation-specific behavior.

`class NSDictionary`

A static collection of objects associated with unique keys.

`class NSMutableDictionary`

A dynamic collection of objects associated with unique keys.

Supporting Types

`struct Keys`

A view of a dictionary’s keys.

`struct Values`

A view of a dictionary’s values.

`struct Index`

The position of a key-value pair in a dictionary.

`typealias Indices`

A type that represents the indices that are valid for subscripting the collection, in ascending order.

`struct Iterator`

An iterator over the members of a `Dictionary<Key, Value>`.

Creating a Dictionary from an Attribute Container

`init<S>(AttributeContainer, including: S.Type) throws` `init<S>(AttributeContainer, including: KeyPath<AttributeScopes, S.Type>) throws`

`init(AttributeContainer)`

Infrequently Used Functionality

`init(dictionaryLiteral: (Key, Value)...)`

Creates a dictionary initialized with a dictionary literal.

`var hashValue: Int`

The hash value.

Type Aliases

`typealias Element`

The element type of a dictionary: a tuple containing an individual key-value pair.

Default Implementations

- Collection Implementations
- CustomDebugStringConvertible Implementations
- CustomReflectable Implementations
- CustomStringConvertible Implementations
- Decodable Implementations
- Encodable Implementations
- Equatable Implementations
- ExpressibleByDictionaryLiteral Implementations
- Hashable Implementations
- Sequence Implementations

Relationships

Conforms To

`CVarArg`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`Collection`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`Copyable`

Conforms when `Key` conforms to `Encodable`, `Key` conforms to `Hashable`, and `Value` conforms to `Encodable`.

`CustomDebugStringConvertible`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`CustomReflectable`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`CustomStringConvertible`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`Decodable`

Conforms when `Key` conforms to `Decodable`, `Key` conforms to `Hashable`, and `Value` conforms to `Decodable`.

`Encodable`

Conforms when `Key` conforms to `Encodable`, `Key` conforms to `Hashable`, and `Value` conforms to `Encodable`.

`Equatable`

Conforms when `Key` conforms to `Hashable` and `Value` conforms to `Equatable`.

`ExpressibleByDictionaryLiteral`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`Hashable`

Conforms when `Key` conforms to `Hashable` and `Value` conforms to `Hashable`.

`MLDataValueConvertible`

`Sendable`

Conforms when `Key` conforms to `Hashable`, `Key` conforms to `Sendable`, `Value` conforms to `Copyable`, `Value` conforms to `Escapable`, and `Value` conforms to `Sendable`.

`SendableMetatype`

Conforms when `Key` conforms to `Hashable`, `Key` conforms to `Sendable`, `Value` conforms to `Copyable`, `Value` conforms to `Escapable`, and `Value` conforms to `Sendable`.

`Sequence`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

See Also

Standard Library

`struct Int`

A signed integer value type.

`struct Double`

A double-precision, floating-point value type.

`struct String`

A Unicode string value that is a collection of characters.

`struct Array`

An ordered, random-access collection.

Swift Standard Library

Solve complex problems and write high-performance, readable code.