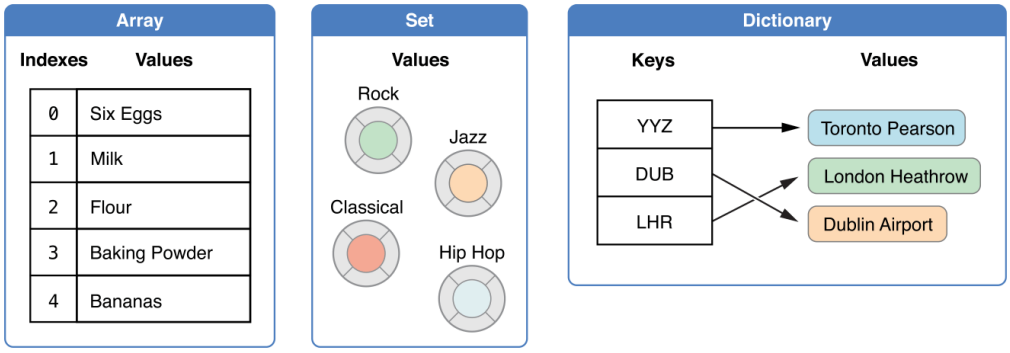
Collection Types

On This Page

Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.



Arrays, sets, and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you cannot insert a value of the wrong type into a collection by mistake. It also means you can be confident about the type of values you will retrieve from a collection.

> NOTE
>
> Swift's array, set, and dictionary types are implemented as *generic collections*. For more about generic types and collections, see Generics.

Mutability of Collections

If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be *mutable*. This means that you can change (or *mutate*) the collection after it's created by adding, removing, or changing items in the collection. If you assign an array, a set, or a dictionary to a constant, that collection is *immutable*, and its size and contents cannot be changed.

> NOTE
>
> It is good practice to create immutable collections in all cases where the collection does not need to change. Doing so makes it easier for you to reason about your code and enables the Swift compiler to optimize the performance of the collections you create.

Arrays

An *array* stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

> NOTE
>
> Swift's `Array` type is bridged to Foundation's `NSArray` class.
>
> For more information about using `Array` with Foundation and Cocoa, see Working with Cocoa Data Types in *Using Swift with Cocoa and Objective-C (Swift 4.1)*.

Array Type Shorthand Syntax

The type of a Swift array is written in full as `Array<Element>`, where `Element` is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as `[Element]`. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of an array.

Creating an Empty Array

You can create an empty array of a certain type using initializer syntax:

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
```

Note that the type of the `someInts` variable is inferred to be `[Int]` from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty array with an empty array literal, which is written as `[]` (an empty pair of square brackets):

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of type [Int]
```

Creating an Array with a Default Value

Swift's `Array` type also provides an initializer for creating an array of a certain size with all of its values set to the same default value. You pass this initializer a default value of the appropriate type (called `repeating`): and the number of times that value is repeated in the new array (called `count`):

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

Creating an Array by Adding Two Arrays Together

You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array's type is inferred from the type of the two arrays you add together:

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

Creating an Array with an Array Literal

You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection. An array literal is written as a list of values, separated by commas, surrounded by a pair of square brackets:

```
[value 1, value 2, value 3]
```

The example below creates an array called `shoppingList` to store `String` values:

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

The `shoppingList` variable is declared as “an array of string values”, written as `[String]`. Because this particular array has specified a value type of `String`, it is allowed to store `String` values only. Here, the `shoppingList` array is initialized with two `String` values (`"Eggs"` and `"Milk"`), written within an array literal.

> NOTE
>
> The `shoppingList` array is declared as a variable (with the `var` introducer) and not a constant (with the `let` introducer) because more items are added to the shopping list in the examples below.

In this case, the array literal contains two `String` values and nothing else. This matches the type of the `shoppingList` variable’s declaration (an array that can only contain `String` values), and so the assignment of the array literal is permitted as a way to initialize `shoppingList` with two initial items.

Thanks to Swift’s type inference, you don’t have to write the type of the array if you’re initializing it with an array literal containing values of the same type. The initialization of `shoppingList` could have been written in a shorter form instead:

```
var shoppingList = ["Eggs", "Milk"]
```

Because all values in the array literal are of the same type, Swift can infer that `[String]` is the correct type to use for the `shoppingList` variable.

Accessing and Modifying an Array

You access and modify an array through its methods and properties, or by using subscript syntax.

To find out the number of items in an array, check its read-only `count` property:

```
print("The shopping list contains \(shoppingList.count) items.")
// Prints "The shopping list contains 2 items."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// Prints "The shopping list is not empty."
```

You can add a new item to the end of an array by calling the array’s `append(_:)` method:

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

Alternatively, append an array of one or more compatible items with the addition assignment operator (+=):

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

Retrieve a value from the array by using *subscript syntax*, passing the index of the value you want to retrieve within square brackets immediately after the name of the array:

```
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
```

> NOTE
> The first item in the array has an index of `0`, not 1. Arrays in Swift are always zero-indexed.

You can use subscript syntax to change an existing value at a given index:

```
shoppingList[0] = "Six eggs"
// the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

When you use subscript syntax, the index you specify needs to be valid. For example, writing `shoppingList[shoppingList.count] = "Salt"` to try to append an item to the end of the array results in a runtime error.

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces `"Chocolate Spread"`, `"Cheese"`, and `"Butter"` with `"Bananas"` and `"Apples"`:

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
```

To insert an item into the array at a specified index, call the array's `insert(_:at:)` method:

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

This call to the `insert(_:at:)` method inserts a new item with a value of `"Maple Syrup"` at the very beginning of the shopping list, indicated by an index of `0`.

Similarly, you remove an item from the array with the `remove(at:)` method. This method removes the item at the specified index and returns the removed item (although you can ignore the returned value if you do not need it):

```
let mapleSyrup = shoppingList.remove(at: 0)
// the item that was at index 0 has just been removed
// shoppingList now contains 6 items, and no Maple Syrup
// the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

> NOTE
> If you try to access or modify a value for an index that is outside of an array's existing bounds, you will trigger a runtime error. You can check that an index is valid before using it by comparing it to the array's `count` property. The largest valid index in an array is `count - 1` because arrays are indexed from zero—however, when `count` is `0` (meaning the array is empty), there are no valid indexes.

Any gaps in an array are closed when an item is removed, and so the value at index `0` is once again equal to `"Six eggs"`:

```
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
```

If you want to remove the final item from an array, use the `removeLast()` method rather than the `remove(at:)` method to avoid the need to query the array's `count` property. Like the `remove(at:)` method, `removeLast()` returns the removed item:

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no apples
// the apples constant is now equal to the removed "Apples" string
```

Iterating Over an Array

You can iterate over the entire set of values in an array with the `for-in` loop:

```
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

If you need the integer index of each item as well as its value, use the `enumerated()` method to iterate over the array instead. For each item in the array, the `enumerated()` method returns a tuple composed of an integer and the item. The integers start at zero and count up by one for each item; if you enumerate over a whole array, these integers match the items' indices. You can decompose the tuple into temporary constants or variables as part of the iteration:

```
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

For more about the `for-in` loop, see For-In Loops.

Sets

A *set* stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

> NOTE
>
> Swift's `Set` type is bridged to Foundation's `NSSet` class.
>
> For more information about using `Set` with Foundation and Cocoa, see Working with Cocoa Data Types in *Using Swift with Cocoa and Objective-C (Swift 4.1)*.

Hash Values for Set Types

A type must be *hashable* in order to be stored in a set—that is, the type must provide a way to compute a

hash value for itself. A hash value is an `Int` value that is the same for all objects that compare equally, such that if `a == b`, it follows that `a.hashValue == b.hashValue`.

All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default, and can be used as set value types or dictionary key types. Enumeration case values without associated values (as described in Enumerations) are also hashable by default.

> NOTE
>
> You can use your own custom types as set value types or dictionary key types by making them conform to the `Hashable` protocol from Swift's standard library. Types that conform to the `Hashable` protocol must provide a gettable `Int` property called `hashValue`. The value returned by a type's `hashValue` property is not required to be the same across different executions of the same program, or in different programs.
>
> Because the `Hashable` protocol conforms to `Equatable`, conforming types must also provide an implementation of the equals operator (==). The `Equatable` protocol requires any conforming implementation of == to be an equivalence relation. That is, an implementation of == must satisfy the following three conditions, for all values `a`, `b`, and `c`:
>
> - `a == a` (Reflexivity)
> - `a == b` implies `b == a` (Symmetry)
> - `a == b && b == c` implies `a == c` (Transitivity)
>
> For more information about conforming to protocols, see Protocols.

Set Type Syntax

The type of a Swift set is written as `Set<Element>`, where `Element` is the type that the set is allowed to store. Unlike arrays, sets do not have an equivalent shorthand form.

Creating and Initializing an Empty Set

You can create an empty set of a certain type using initializer syntax:

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count) items.")
// Prints "letters is of type Set<Character> with 0 items."
```

> NOTE
>
> The type of the `letters` variable is inferred to be `Set<Character>`, from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty set with an empty array literal:

```
letters.insert("a")
// letters now contains 1 value of type Character
letters = []
// letters is now an empty set, but is still of type Set<Character>
```

Creating a Set with an Array Literal

You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection.

The example below creates a set called `favoriteGenres` to store `String` values:

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

The `favoriteGenres` variable is declared as "a set of `String` values", written as `Set<String>`. Because this particular set has specified a value type of `String`, it is *only* allowed to store `String` values. Here, the `favoriteGenres` set is initialized with three `String` values (`"Rock"`, `"Classical"`, and `"Hip hop"`), written within an array literal.

> NOTE
>
> The `favoriteGenres` set is declared as a variable (with the `var` introducer) and not a constant (with the `let` introducer) because items are added and removed in the examples below.

A set type cannot be inferred from an array literal alone, so the type `Set` must be explicitly declared. However, because of Swift's type inference, you don't have to write the type of the set if you're initializing it with an array literal containing values of the same type. The initialization of `favoriteGenres` could have been written in a shorter form instead:

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

Because all values in the array literal are of the same type, Swift can infer that `Set<String>` is the correct type to use for the `favoriteGenres` variable.

Accessing and Modifying a Set

You access and modify a set through its methods and properties.

To find out the number of items in a set, check its read-only `count` property:

```
print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}
// Prints "I have particular music preferences."
```

You can add a new item into a set by calling the set's `insert(_:)` method:

```
favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
```

You can remove an item from a set by calling the set's `remove(_:)` method, which removes the item if it's a member of the set, and returns the removed value, or returns `nil` if the set did not contain it. Alternatively, all items in a set can be removed with its `removeAll()` method.

```
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."
```

To check whether a set contains a particular item, use the `contains(_:)` method.

```
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."
```

Iterating Over a Set

You can iterate over the values in a set with a `for-in` loop.

```
for genre in favoriteGenres {
    print("\(genre)")
}
// Jazz
// Hip hop
// Classical
```

For more about the `for-in` loop, see For-In Loops.

Swift's `Set` type does not have a defined ordering. To iterate over the values of a set in a specific order, use the `sorted()` method, which returns the set's elements as an array sorted using the < operator.

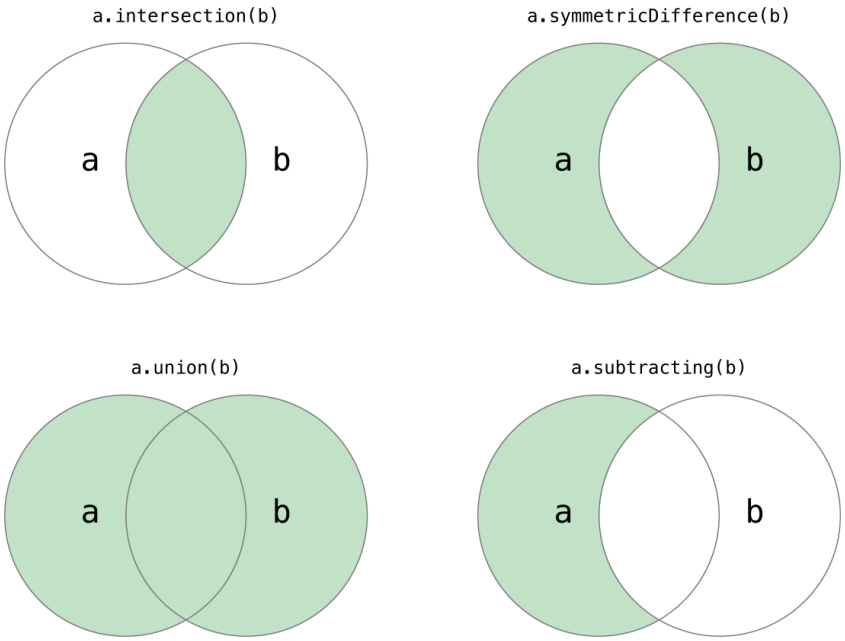
```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

Performing Set Operations

You can efficiently perform fundamental set operations, such as combining two sets together, determining which values two sets have in common, or determining whether two sets contain all, some, or none of the same values.

Fundamental Set Operations

The illustration below depicts two sets—`a` and `b`—with the results of various set operations represented by the shaded regions.



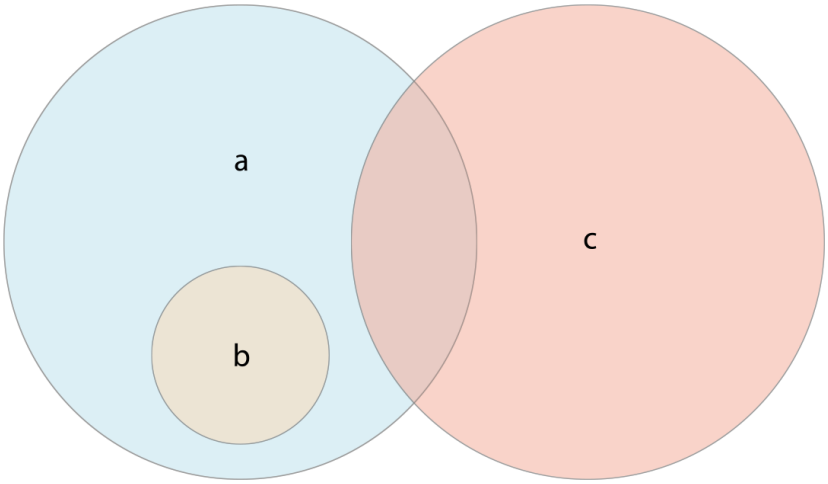
- Use the `intersection(_:)` method to create a new set with only the values common to both sets.
- Use the `symmetricDifference(_:)` method to create a new set with values in either set, but not both.
- Use the `union(_:)` method to create a new set with all of the values in both sets.
- Use the `subtracting(_:)` method to create a new set with values not in the specified set.

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Set Membership and Equality

The illustration below depicts three sets—`a`, `b` and `c`—with overlapping regions representing elements shared among sets. Set `a` is a *superset* of set `b`, because `a` contains all elements in `b`. Conversely, set `b` is a *subset* of set `a`, because all elements in `b` are also contained by `a`. Set `b` and set `c` are *disjoint* with one another, because they share no elements in common.



- Use the “is equal” operator (==) to determine whether two sets contain all of the same values.
- Use the `isSubset(of:)` method to determine whether all of the values of a set are contained in the specified set.
- Use the `isSuperset(of:)` method to determine whether a set contains all of the values in a specified set.
- Use the `isStrictSubset(of:)` or `isStrictSuperset(of:)` methods to determine whether a set is a subset or superset, but not equal to, a specified set.
- Use the `isDisjoint(with:)` method to determine whether two sets have no values in common.

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

Dictionaries

A *dictionary* stores associations between keys of the same type and values of the same type in a collection with no defined ordering. Each value is associated with a unique *key*, which acts as an identifier for that value within the dictionary. Unlike items in an array, items in a dictionary do not have a specified order. You use a dictionary when you need to look up values based on their identifier, in much the same way that a real-world dictionary is used to look up the definition for a particular word.

> NOTE
>
> Swift’s `Dictionary` type is bridged to Foundation’s `NSDictionary` class.
>
> For more information about using `Dictionary` with Foundation and Cocoa, see Working with Cocoa Data Types in *Using Swift with Cocoa and Objective-C (Swift 4.1)*.

Dictionary Type Shorthand Syntax

The type of a Swift dictionary is written in full as `Dictionary<Key, Value>`, where `Key` is the type of value that

can be used as a dictionary key, and `Value` is the type of value that the dictionary stores for those keys.

> NOTE
> A dictionary `Key` type must conform to the `Hashable` protocol, like a set's value type.

You can also write the type of a dictionary in shorthand form as `[Key: Value]`. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of a dictionary.

Creating an Empty Dictionary

As with arrays, you can create an empty `Dictionary` of a certain type by using initializer syntax:

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty [Int: String] dictionary
```

This example creates an empty dictionary of type `[Int: String]` to store human-readable names of integer values. Its keys are of type `Int`, and its values are of type `String`.

If the context already provides type information, you can create an empty dictionary with an empty dictionary literal, which is written as `[:]` (a colon inside a pair of square brackets):

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
namesOfIntegers = [:]
// namesOfIntegers is once again an empty dictionary of type [Int: String]
```

Creating a Dictionary with a Dictionary Literal

You can also initialize a dictionary with a *dictionary literal*, which has a similar syntax to the array literal seen earlier. A dictionary literal is a shorthand way to write one or more key-value pairs as a `Dictionary` collection.

A *key-value pair* is a combination of a key and a value. In a dictionary literal, the key and value in each key-value pair are separated by a colon. The key-value pairs are written as a list, separated by commas, surrounded by a pair of square brackets:

```
[key 1: value 1, key 2: value 2, key 3: value 3]
```

The example below creates a dictionary to store the names of international airports. In this dictionary, the keys are three-letter International Air Transport Association codes, and the values are airport names:

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

The `airports` dictionary is declared as having a type of `[String: String]`, which means "a `Dictionary` whose keys are of type `String`, and whose values are also of type `String`".

> NOTE
> The `airports` dictionary is declared as a variable (with the `var` introducer), and not a constant (with the `let` introducer), because more airports are added to the dictionary in the examples below.

The `airports` dictionary is initialized with a dictionary literal containing two key-value pairs. The first pair has a key of `"YYZ"` and a value of `"Toronto Pearson"`. The second pair has a key of `"DUB"` and a value of `"Dublin"`.

This dictionary literal contains two `String: String` pairs. This key-value type matches the type of the

`airports` variable declaration (a dictionary with only `String` keys, and only `String` values), and so the assignment of the dictionary literal is permitted as a way to initialize the `airports` dictionary with two initial items.

As with arrays, you don't have to write the type of the dictionary if you're initializing it with a dictionary literal whose keys and values have consistent types. The initialization of `airports` could have been written in a shorter form instead:

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Because all keys in the literal are of the same type as each other, and likewise all values are of the same type as each other, Swift can infer that `[String: String]` is the correct type to use for the `airports` dictionary.

Accessing and Modifying a Dictionary

You access and modify a dictionary through its methods and properties, or by using subscript syntax.

As with an array, you find out the number of items in a `Dictionary` by checking its read-only `count` property:

```
print("The airports dictionary contains \(airports.count) items.")
// Prints "The airports dictionary contains 2 items."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}
// Prints "The airports dictionary is not empty."
```

You can add a new item to a dictionary with subscript syntax. Use a new key of the appropriate type as the subscript index, and assign a new value of the appropriate type:

```
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
```

You can also use subscript syntax to change the value associated with a particular key:

```
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London Heathrow"
```

As an alternative to subscripting, use a dictionary's `updateValue(_:forKey:)` method to set or update the value for a particular key. Like the subscript examples above, the `updateValue(_:forKey:)` method sets a value for a key if none exists, or updates the value if that key already exists. Unlike a subscript, however, the `updateValue(_:forKey:)` method returns the *old* value after performing an update. This enables you to check whether or not an update took place.

The `updateValue(_:forKey:)` method returns an optional value of the dictionary's value type. For a dictionary that stores `String` values, for example, the method returns a value of type `String?`, or "optional `String`". This optional value contains the old value for that key if one existed before the update, or `nil` if no value existed:

```
if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
// Prints "The old value for DUB was Dublin."
```

You can also use subscript syntax to retrieve a value from the dictionary for a particular key. Because it is

possible to request a key for which no value exists, a dictionary's subscript returns an optional value of the dictionary's value type. If the dictionary contains a value for the requested key, the subscript returns an optional value containing the existing value for that key. Otherwise, the subscript returns `nil`:

```
if let airportName = airports["DUB"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport is not in the airports dictionary.")
}
// Prints "The name of the airport is Dublin Airport."
```

You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of `nil` for that key:

```
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

Alternatively, remove a key-value pair from a dictionary with the `removeValue(forKey:)` method. This method removes the key-value pair if it exists and returns the removed value, or returns `nil` if no value existed:

```
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}
// Prints "The removed airport's name is Dublin Airport."
```

Iterating Over a Dictionary

You can iterate over the key-value pairs in a dictionary with a `for-in` loop. Each item in the dictionary is returned as a `(key, value)` tuple, and you can decompose the tuple's members into temporary constants or variables as part of the iteration:

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
```

For more about the `for-in` loop, see For-In Loops.

You can also retrieve an iterable collection of a dictionary's keys or values by accessing its `keys` and `values` properties:

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
// Airport code: YYZ
// Airport code: LHR

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: Toronto Pearson
```

```
// Airport name: London Heathrow
```

If you need to use a dictionary's keys or values with an API that takes an `Array` instance, initialize a new array with the `keys` or `values` property:

```
let airportCodes = [String](airports.keys)
// airportCodes is ["YYZ", "LHR"]

let airportNames = [String](airports.values)
// airportNames is ["Toronto Pearson", "London Heathrow"]
```

Swift's `Dictionary` type does not have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the `sorted()` method on its `keys` or `values` property.