```swift
import UIKit
//: # Collection Types
/*:
 * Arrays : ordered collections of values
 * Set : unordered collections of unique values
 * Dictionary : unordered collections of key-value associations
*/
//: ## Arrays
//: ### An array stores values of the same type in an ordered list
//: ### Declaration:
//:     Array<Element>
//:     [Element]
//: ### Creating an Empty Array
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
//: ### if the context already provides type information
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
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]

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
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
/*:
 ### Thanks to Type Inference:
 * var shoppingList = ["Eggs", "Milk"]
*/
//: ### Accessing and Modifying an Array
print("The shopping list contains \(shoppingList.count) items.")
// Prints "The shopping list contains 2 items."

if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// Prints "The shopping list is not empty."
//: ### Add a new item to the end of an Array -> append(_:)
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

```swift
//: ### append an array of one or more compatible items
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
//: ### Retrieve a value from the array
//: * subscript syntax
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
//: ### 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"
//: ### Note: the index you specify needs to be valid
//: * shoppingList[shoppingList.count] = "Salt" --> runtime error
//: ### change a range of values at once
//: Let's see what is inside shoppingList
shoppingList.count
print(shoppingList)

shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
print(shoppingList)
//: ### insert an item into the array at a specified index
//: * insert(_:at:)
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
//: ### remove an item from the array
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
//: ### Any gaps in an array are closed when an item is removed
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
//: ### remove the final item from an array -->removeLast()
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
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
//: ### If you need the integer index of each item --> enumerated() method
//: ### the enumerated() method returns a tuple composed of an integer and the
 item
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
//: ## Sets
//: * Set<Element>
//: ### 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
//: ### Creating and Initializing an Empty Set
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."
//: ### if the context already provides type information, 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
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
//: ### Note: A set type cannot be inferred from an array literal alone, so
 the type Set must be explicitly declared!!!
//: * if you’re initializing it with an array literal containing values of the
 same type:
//: * var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
//: ### Accessing and Modifying a Set
print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."
//: * Boolean isEmpty property:
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."
//: ### add a new item into a set --> insert(_:)
favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
//: ### remove an item from a set
//: * if the member exist in the set, it eturns the item
//: * if not it returns nil
//: * 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."
//: ### check whether a set contains a particular item --> contains(_:)
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
for genre in favoriteGenres {
    print("\(genre)")
}
// Classical
// Jazz
// Hip hop
//: ### To iterate over the values of a set in a specific order --> sorted()
//: ### they will be in the ascending order
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
//: ### To sort the elements of your sequence in descending order -->
 sorted(by: >)
for genre in favoriteGenres.sorted(by: >) {
    print("\(genre)")
}
// Jazz
// Hip hop
// Classical
//: ## Performing Set Operations
//: ### Fundamental Set Operations
//: ### intersection(_:) --> create a new set with only the values common to
 both sets
//: ### symmetricDifference(_:) --> create a new set with values in either
 set, but not both
//: ### union(_:) --> create a new set with all of the values in both sets
//: ### subtracting(_:) --> 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]
```

```swift
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
//: ### Set Membership and Equality
//: ### "is equal" operator (==) --> determine whether two sets contain all of
 the same values
//: ### isSubset(of:) --> determine whether all of the values of a set are
 contained in the specified set
//: ### isSuperset(of:) --> determine whether a set contains all of the values
 in a specified set
//: ### isStrictSubset(of:) or isStrictSuperset(of:) --> determine whether a
 set is a subset or superset, but not equal to, a specified set
//: ### isDisjoint(with:) --> 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
 * items in a dictionary do not have a specified order
 * Dictionary<Key, Value> --> Key is the type of value, Value is the type of
  value
 * shorthand form (prefered) --> [Key: Value]
*/
//: ### Creating an Empty Dictionary
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty [Int: String] dictionary
// Its keys are of type Int, and its values are of type String
//: ### If the context already provides type information --> [:]
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
print(namesOfIntegers)
namesOfIntegers = [:]
// namesOfIntegers is once again an empty dictionary of type [Int: String]
print(namesOfIntegers)
//: ### Creating a Dictionary with a Dictionary Literal
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

```
// the keys are three-letter International Air Transport Association codes,
 and the values are airport names
// airports dictionary is declared as [String: String]
//: ### Since we are initializing the aitports dictionary with a dictionary
 literal
// var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"] --> [String:
 String] is infered
//: ### Accessing and Modifying a Dictionary
//: ### find out the number of items in a Dictionary:
print("The airports dictionary contains \(airports.count) items.")
// Prints "The airports dictionary contains 2 items."
//: ### isEmpty --> check 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."
//: ### add a new item to a dictionary:
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
//: ### change the value associated with a particular key:
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London Heathrow"
//: ### Alternative way --> use updateValue(_:forKey:) method
//: * updateValue(_:forKey:) method sets a value for a key if none exists, or
 updates the value if that key already exists
//: * updateValue(_:forKey:) method returns the old value after performing an
 update
//: * The updateValue(_:forKey:) method returns an optional value of the
 dictionary’s value type
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."
//: ### using subscript syntax to retrieve a value from the dictionary for a
 particular key
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."
//: ### remove a key-value pair from a dictionary by assigning a value of nil
 for that key:
print(airports)
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so delete it
print(airports)
airports["APL"] = nil
// APL has now been removed from the dictionary
```

```
print(airports)
//: ### Alternative way --> remove a key-value pair from a dictionary with the
 removeValue(forKey:) method
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
//: * Each item in the dictionary is returned as a (key, value) tuple
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
//: ### 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
//: ### initializing 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"]
//: ### Sort a Dictionary upon the key or the values --> sorted() method
print(airports.keys.sorted())
print(airports.values.sorted())
```