```
import UIKit
import Foundation

/*:
 # Strings and Characters
 Strings Are Value Types

 if you import Foundation, you can access those NSString methods on String
  without casting
 *    import Foundation
*/
//: ## String Literals
let someString = "Some string literal value"
//: ## Multiline String Literals
//: * three double quotation marks
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
//: ### If you want to use line breaks to make your source code easier to
 read, but you don't want the line breaks to be part of the string's value,
 write a backslash (\) at the end of those lines:
let softWrappedQuotation = """
The White Rabbit put on his spectacles.  "Where shall I begin, \
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on \
till you come to the end; then stop."
"""
//: ## Special Characters in String Literals
/*:
 ### The escaped special characters
 * \0 (null character)
 * \\ (backslash)
 * \t (horizontal tab)
 * \n (line feed)
 * \r (carriage return)
 * \" (double quotation mark) and \' (single quotation mark)
 * An arbitrary Unicode scalar, written as \u{n}
*/
//: ### Example:
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665
let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
//: ### double quotation mark (") can be used inside of a multiline string
 literal without escaping it
```

```
let threeDoubleQuotationMarks = """
Escaping the first quotation mark \"""
Escaping all three quotation marks \"\"\"
"""
//: ## Initializing an Empty String
var emptyString = ""               // empty string literal
var anotherEmptyString = String()  // initializer syntax
// these two strings are both empty, and are equivalent to each other
//: ### isEmpty property
if emptyString.isEmpty {
    print("Nothing to see here")
}
// Prints "Nothing to see here"
//: ## String Mutability
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"

let constantString = "Highlander"
//constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
//: ## Working with Characters
//: ### Iterating over the string with a for-in loop:
for character in "Dog!🐶" {
    print(character)
}
// D
// o
// g
// !
// 🐶
//: ### Character type annotation:
let exclamationMark: Character = "!"
//: ### passing an array of Character values as an argument to its initializer:
let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]
let catString = String(catCharacters)
print(catString)
// Prints "Cat!🐱"
//: ## Concatenating Strings and Characters
//: * addition operator (+)
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
// welcome now equals "hello there"
//: * addition assignment operator (+=):
var instruction = "look over"
instruction += string2
// instruction now equals "look over there"
//: * append a Character or String value to a String variable
welcome.append(exclamationMark)
```

```
// welcome now equals "hello there!"
//: ## String Interpolation
//: * Each item is wrapped in a pair of parentheses, prefixed by a backslash
 (\):
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
//: ## Unicode
//: ### Unicode is an international standard for encoding, representing, and
 processing text in different writing systems.
let chick = "\u{1F425}"
//: ### Extended Grapheme Clusters
//: * An extended grapheme cluster is a sequence of one or more Unicode
 scalars that (when combined) produce a single human-readable character.
let eAcute: Character = "\u{E9}"                          // é
let combinedEAcute: Character = "\u{65}\u{301}"           // e followed by
// eAcute is é, combinedEAcute is é

let enclosedEAcute: Character = "\u{E9}\u{20DD}"
// enclosedEAcute is ⓔ⃝

let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"
// regionalIndicatorForUS is 🇺🇸

//: ## Counting Characters
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
print("unusualMenagerie has \(unusualMenagerie.count) characters")
// Prints "unusualMenagerie has 40 characters"
//: ### Note!!!
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

word += "\u{301}"    // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
//: ## Accessing and Modifying a String
/*:
 ### String Indices
 * **startIndex** property -> access the position of the first Character of a
  String
 * **endIndex** property -> the position after the last character in a String
    * If a String is empty, startIndex and endIndex are equal
 * **index(before:)**
 * **index(after:)**
 * **index(_:offsetBy:)**
*/
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
```

```
// G
greeting[greeting.index(before: greeting.endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a
greeting[greeting.index(greeting.index(after: greeting.startIndex), offsetBy:
 2)]
// e
//: ### indices property
for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
// Prints "G u t e n   T a g ! "
/*:
 ### Inserting and Removing
 ### Insert
 * insert(_:at:)
 * insert(contentsOf:at:)
*/
var welcomeAgain = "hello"
welcomeAgain.insert("!", at: welcomeAgain.endIndex)
// welcomeAgain now equals "hello!"

welcomeAgain.insert(contentsOf: " there", at: welcomeAgain.index(before:
 welcomeAgain.endIndex))
// welcomeAgain now equals "hello there!"
/*:
 ### Remove
 * remove(at:)
 * removeSubrange(_:)
*/
welcomeAgain.remove(at: welcomeAgain.index(before: welcomeAgain.endIndex))
// welcomeAgain now equals "hello there"

let range = welcomeAgain.index(welcomeAgain.endIndex, offsetBy:
 -6)..<welcomeAgain.endIndex
welcomeAgain.removeSubrange(range)
// welcomeAgain now equals "hello"
//: ## Substrings
//: * substrings aren't suitable for long-term storage
//: * as a performance optimization, a substring can reuse part of the memory
 that's used to store the original string, or part of the memory that's used
 to store another substring
let greeting2 = "Hello, world!"
greeting2.first
let index2 = greeting2.index(of: ",") ?? greeting2.endIndex
print(index2)
let ii = greeting2.firstIndex(of: ",") ?? greeting2.endIndex
```

```
print(ii)
let beginning = greeting2[..<index2]
let beg = greeting2[..<ii]
// beginning is "Hello"

// Convert the result to a String for long-term storage.
let newString = String(beginning)
//: ### prefix(_:)
greeting2.prefix(5)
//: ## Comparing Strings
//: ### String and Character Equality
//: * the “equal to” operator (==)
//: * the “not equal to” operator (!=)
let aQuotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if aQuotation == sameQuotation {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
//: ### Two String values (or two Character values) are considered equal if
 their extended grapheme clusters are canonically equivalent in other words if
 they have the same linguistic meaning and appearance
// "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
//: ### but these two don’t have the same linguistic meaning:
let latinCapitalLetterA: Character = "\u{41}"

let cyrillicCapitalLetterA: Character = "\u{0410}"

if latinCapitalLetterA != cyrillicCapitalLetterA {
    print("These two characters are not equivalent.")
}
// Prints "These two characters are not equivalent."
//: ### lowercase and uppercase
var newGreeting = greeting2.uppercased()
newGreeting.lowercased()
newGreeting.capitalized




var ss = "Hello"
print(ss[ss.startIndex])
```

```
print(ss[ss.index(before: ss.endIndex)])

print(ss[ss.index(after: ss.startIndex)])
print(ss[ss.index(ss.index(after: ss.startIndex), offsetBy: 3)])
```