import UIKit  
//: # Closures  
//: ### Closures are self-contained blocks of functionality that can be passed around and used in your code  
//: ### Closures take one of three forms:  
//: \* Global functions are closures that have a name and do not capture any values.  
//: \* Nested functions are closures that have a name and can capture values from their enclosing function.  
//: \* Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.  
//: ## Closure Expressions  
//: ### The sorted method  
let names : Set = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]  
func backward(\_ s1: String, \_ s2: String) -> Bool {  
 return s1 > s2  
}  
var reversedNames = names.sorted(by: backward)  
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]  
/\*:  
 ### Closure Expression Syntax:  
 \*{ (parameters) -> return type in\*  
  
 \*\/\/statements\*  
   
 \*}\*  
\*/  
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in  
 return s1 > s2  
})  
//: ### Inferring Type From Context  
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )  
//: ### Implicit Returns from Single-Expression Closures  
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )  
//: ### Shorthand Argument Names  
reversedNames = names.sorted(by: { $0 > $1 } )  
//: ### Operator Methods  
//: string-specific implementation of the greater-than operator (>)  
reversedNames = names.sorted(by: >)  
//: ### Trailing Closures  
reversedNames = names.sorted() { $0 > $1 }  
reversedNames = names.sorted { $0 > $1 }  
//: ## Example:  
//: ### passing a function as an argument to another function  
func calculator (n1: Int, n2: Int, operation: (Int, Int) -> Int) -> Int {  
 return operation(n1, n2)  
}  
func add (no1: Int, no2: Int) -> Int {  
 return no1 + no2  
}  
func multiply (no1: Int, no2: Int) -> Int {  
 return no1 \* no2  
}  
calculator(n1: 2, n2: 3, operation: add)  
calculator(n1: 2, n2: 3, operation: multiply)  
calculator(n1: 2, n2: 3, operation: {(no1: Int, no2: Int) -> Int in  
 return no1 \* no2  
})  
//: ### Inferring Type From Context  
calculator(n1: 2, n2: 3, operation: {(no1, no2) -> Int in  
 return no1 \* no2  
})  
//: even further  
calculator(n1: 2, n2: 3, operation: {(no1, no2) in  
 return no1 \* no2  
})  
//: ### Implicit Returns from Single-Expression Closures  
calculator(n1: 2, n2: 3, operation: {(no1, no2) in no1 \* no2})  
//: ### Shorthand Argument Names  
//: ### Anonymous parameter names  
let result = calculator(n1: 2, n2: 3, operation: {$0 \* $1})  
print(result)  
//: ### Trailing Closures  
//: ### if the last parameter in your function is a closure, we can omit the parameter name and close the parantheses before the closure  
let result1 = calculator(n1: 2, n2: 3) {$0 \* $1}  
print(result1)  
//: ## Example 2:  
let array = [6,2,3,9,4,1]  
//: ### we want add 1 to everey item of the array  
func addOne (n1: Int) -> Int {  
 return n1 + 1  
}  
array.map(addOne)  
//: ### making a closure  
let res1 = array.map({(n1) in n1 + 1})  
let res2 = array.map {$0 + 1}  
//let res2 = array.map({$0 + 1})  
print(res2)  
//: ## Capturing Values  
//: \* A closure can capture constants and variables from the surrounding context in which it is defined  
//: \* The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists  
//: ### Example:  
func makeIncrementer(forIncrement amount: Int) -> () -> Int {  
 var runningTotal = 0  
 func incrementer() -> Int {  
 runningTotal += amount  
 return runningTotal  
 }  
 return incrementer  
}  
let incrementByTen = makeIncrementer(forIncrement: 10)  
incrementByTen()  
// returns a value of 10  
incrementByTen()  
// returns a value of 20  
incrementByTen()  
// returns a value of 30  
  
let incrementBySeven = makeIncrementer(forIncrement: 7)  
incrementBySeven()  
// returns a value of 7  
//: ### If we call again incrementByTen, it increments its own runningTotal variable, and does not affect the variable captured by incrementBySeven:  
incrementByTen()  
// returns a value of 40  
//: ## Closures Are Reference Types  
//: ### functions and closures are reference types  
//: ### Whenever you assign a function or a closure to a constant or a variable, you are actually setting that constant or variable to be a reference to the function or closure  
//: ### In the example above, it is the choice of closure that incrementByTen refers to that is constant, and not the contents of the closure itself  
let alsoIncrementByTen = incrementByTen  
alsoIncrementByTen()  
// returns a value of 50