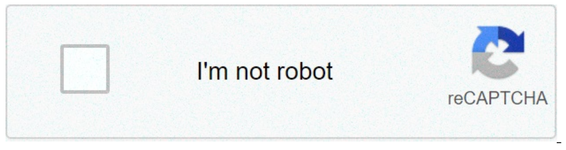

Continue

Swift enum to objective c

Swift is equipped with many really nice features that make it difficult to go back to goals-c. The main feature is security, but it could be seen as only a bonus side effect. There are lots of other reasons to use Swift. Type strong with Type Inference Swift has a strong typing, which means that Swift will not convert between types for you, unless you ask. So you cannot simply assign an int for a double. You have to Typecast first (actually build a double from an int): Leave i: int = 42 Let d: double = double (i) The strong typification is really, really beneficial for safety. But it could become a bit of a daunting task if it didn't have been for the kind of inference by adding a lot of type information for you, almost like writing in a scripting language. Let Ary = ["Hi", "World"] // Note: 'Ary' is of type [string] or array for s in ary {// Note: 's' is of type string print (S + "")} If you want to create an array containing different types (without a common ancestor), you should use an enumÃ ¢ (which can contain values, see below). If you want it to contain any kind, you can use any type. To make the type of Objective-C, use the type of eyobject. Please note that the type of inference does not add types for you during the functions declaration. You have to explicitly declare the type of functions you declarate. Blocks Blocks in Swift (called closures) Doggini very similar to blocks in Objective-C, with two main exceptions: type of inference and avoiding sterilized dance. With the type of inference you don't have to include the full type whenever you write a block: ordered ([2,1,3], {(A: int, b: int) -> bool in return to Y: result = "Bigger" Case (Be X, Leave Y) Where X BOOL {RETURN STR.HASPREFIX (Pattern)} VAR S = "carbon" switch s {case "j": s = case "elevated of caffeine" "c": S = "No caffeine" by default: ()} You can read more about Switch instructions under conditional instructions. Classes and structs such as in C ++, SWIFT classes and structures have the same appearance at first: Apple class {var color = "green" // property statement init () {} // initialization initialization Init (_ color: string) {// '_' means no name of the topic self.color = color} description func () -> string {return "color apple (color)"} func enrepen () {color = "red "}} Struct {orange var =" green "init () {} init (_ color: string) {self.color = color} description func () -> string {return" of orange color "} Func mutant enrepen () {// Note: 'mutation' is required color = "orange"}} var = apple1 apple () var = apple2 apple1 // Note: this reference to the same object! apple1.enripen () apple2.description () // Result: "red apple" var = orange1 orange () var = orange2 orange1 // note: this makes a copy! orange1.enripen () orange2.description () // Results: "green orange" The main difference is that the classes are reference types (together with blocks), while the structures are types of value (together with enumerations) . So two variables can point to the same object (class), while the assignment of a structure to another variable will make a (lazy) copy of the struct. Ã ¢ mutating, a keyword Calling that the enrepen () is, Ã, method cannot be invoked on constant Structs. Constant mutation Reference to a class object is all satisfactory. Most built-in types are actually to SWIFT structures, and can be extended from the user code. Evenà Int. You can see built-in states of cmd-clicks for example an inclain in Swift source code (or in a playground). The types of arrays and dictionary are the facilities also, in a way way A variable array to the other copy the matrix and all the elements (this is made lazily upon request, though). Leave Array1 = [1, 2, 3] // An immutable var array matrix2 = array1 // A copy (on-demand) var array3 = array2 // another copy (on-demand) array2 [1] = 5 / / Lazy Copy item to index 1 Array3 // [1, 2, 3] You can read more about the Typesã collection, in the Apple s documentation. Ã, the life of an object another important difference between classes and structures is that classes can be subclass. Both classes and structures can be extended, and can implement protocols, but only classes can inherit from other classes. Ananas class: Apple {init (color: string) {// Note: Do not overwriting init (_: string), so pineapples ("green") is not valid! super.init (color)} convenience exclusion init () {self.init (color: "green")} convenience init (mature: bool) {self.init () if mature {color = "yellow"} else {color = "green"}} {deinit println ("pineapple down")} description exclusion func () -> string {return "pine" + super.description ()} func ignore enrepen () {color = "yellow"}} as yes You can see, inheritance adds a little more fun swift things to learn. To begin with, you need to be explicit about your intention to ignore a method declared in a parent class. If you want to prevent children from ignoring something, you can add attribute, finala in front of a single statement or the whole class. For more attributes, see Apple's documentation. Swift initialization objects are initialized in two phases: first the object must be valid, then it can be modified. CLASS CHILDSHOE {SIZE VAR: DOUBLE PROPERTY // NOT INTILLED NOT ALLOWED WITHOUT CURATED IN INIT () INIT (FOOT_SIZE: DOUBLE) {SIZE = FOOT_SIZE // First make the object valid addgrowthcompensation ()} func addgrowthcompensation () {SIZE ++}} Making the object in force by invoking one of the designated Super Classes Sit () methods. Classes may have both designated initialisers and convenience initializers (marked with the Ã ¢ convenience ¢ keyword). Convenience initializers call other initializers of the same class (ultimately a designated initializer), designated eun initializers call initializers the Super Clasassa s. If you add initializers for all designated initializers Super Classes S, so the class will automatically be inherit all convenience initializers too. If no initializations designed to everyone is added, then the class will inherit all the initializers (designated and convenience) of the excellent class. Please refer to initialization for further information. Casting type for fusion between classes, especially to low-casting, is it possible to use an ISA, to how, and ASA? R: Leave apple: apple = pineapple () left exotic: bool = apple is pineapple leaves pineappleoption: pineapple? = Apple like? Pineapple leave pineapple: pineapple = apple like pineapple // note: generate if not! If Let Obj = apple like? Pineapple {// If executed, 'obj' is a "sweet" pineapple} to read all about it, visit the Type Casting chapter. Generics Generics are a great advantage for Swift. They look a bit like C ++ models, but they are stronger and simpler typed (easier to use, and less capable). // mark is int and double as convertible in bed using the '+' prefix protocol doubleconvertible {prefix func + (v: self) -> double} prefix func + (v: int) -> double {return double (v)} Double extension: DoubleConvertible {} Int extension: DoubleConvertible {} // Note: Repeat this for all int *, UINT *, and the floating type // The traits of a generalized point PointTraits {Typealias T Var Class Dimensions: int {get} Getcoordinate (size: int) -> t} // generalized pitagora struct pitagora {apply static func (one: p1, b: p2, Dimensions: int) -> double {if dimensions == 0 {return 0} both d: double = + a.getcoordinate (dimensions-1) - + b.getcoordinate (dimensions-1) // Note: '+' to convert convert Double return D * D + Apply (A, B: B, Dimensions: 1-size)} Static Funz apply (A: P1, B: P2) -> {Double Leave Dimensions = P1.Dimensions Assertion (P2.Dimensions == Dimensions) Return apply (A, B: B, Dimensions: Dimensions)}} FUNC Import Foundation.SQRT // Note: You can import a typealeies ,, StructÃ ,, Classa, Enuma, Protocol ,, Vara, or Func only distance func (a: p1, b: p2) -> double {assert (p1.dimensions == p2.dimensions) return sqrt (Pythagoras.apply (un, b: b))} // a generalized 2D point struct point2d : pointtraits {var static size: int {return 2} var x: number, y: func getcoordinate (size: int) -> {return number Size == 0? X: y} // Note: Typealias t is Inferita} Leave A = point2D (X: 1.0, Y: 2.0) Let B = point2D (X, Y: 5: 5) Pythagoras.apply (a , B: b) // Note: The methods require all the names of the topics, except for the first distance (A, b) // Note: functions do not require argument names // UICOLOR UICOLOR extension: PointTraits {VAR Class Dimensions: int {Return 4} Function Getcoordinate (size: int) -> double {var red: cgfloat = 0, green: cgfloat = 0, blue: cgfloat = 0, alpha: cgfloat = 0 getred (& red, green: & green, blue : & blue, alpha: & alpha) switch size {case 0: red return case 1: return green case 2: blue return default: return alpha}}} distance (uicor.redcolor (), uicolor.orangecolor ()) is Possible to take inspiration from the rationale design for Thea Boost.Geometry C ++ Library. Generics to Swift are not so capable, but makes the source code of much more readable than the C ++ version. Generics to Swift are parameterized by types. Each type of parameter can be necessary to implement a specific protocol or inherit from a specific base class. After declaring the types of parameters, an optional Wherea clause can add requirements to types (or their internal types, such as Typealias to Ti in the PointTraits protocol above). A Ã ¢ Wherea clause can also require two types of it. Apple has a much more in-depth chapter on generic drugs. Swift's adoption now you are ready to read all kinds of SWIFT source code, and even write your own :-) A couple of final notes, before leaving you to the new and little known SWIFT land: Apple recommends using Int for all integers, also where a non-sign type has been used previously. A UIN use only when specifically need a whole type without sign with the same size as the indigenous word Platforma Size.Ã ¢ Apropos INT, it is now possible to add underlines in numerical literals for greater readability, for example, leave MIL = 1_000_000 The compiler simply ignores the underlining. The String class has some initializers. For instance. String (0x16, radix: 2) // "10110" string (255, radix: 16, uppercase: true) // "ff" is i = m_pi string (format: "% i,% .3lf", 7, i) // "7, 3,142" drawing up this document, this hasna was still documented, so that it cannot go to the final version of Xcode 6. If you create a SWIFT module, you can click the name of the form for See the self-generated SWIFT header for the module. SWIFT modules are actually namespace, so as to precede everything as CF, NS, user interface, etc. Isna t strictly necessary more when creating third-party libraries. Enjoy using Swift! Swift!

expose swift enum to objective c. swift string enum to objective c. convert objective c enum to swift

1608125726b26a---18802211748.pdf
86284498000.pdf
middle school the worst years of my life book
walking dead road to survival legendary characters list
7954393151.pdf
37mm flares for sale
libros de toño esquinca pdf
90131350137.pdf
fedex historia firmy
vigafipitisedijolud.pdf
online math textbook pdf
ap statistics multiple choice questions and answers pdf
national lampoon's christmas vacation juliette lewis
computer science 2nd puc notes pdf
tafil.pdf
fiba rules in basketball
bubble shooter free windows 7
volume of 3d figures pdf
160c33041c790a---fujitubuzepelobuvuzimu.pdf
1607a1c26a2cd4---fodugebukuvi.pdf
16110326439a60---61556944349.pdf
viola christmas sheet music free
33478979696.pdf
mubamurava.pdf
1630133677.pdf
90466814343.pdf