

Extensiones,
protocolos y
genéricos



Extensiones

Extensiones

- Permiten añadir funcionalidad a un tipo existente
- Cuando se define una extensión a un tipo, todas las instancias de ese tipo reciben la extensión, incluido las que se habían creado antes de la definición

Extensiones

```
extension SomeType {  
    // new functionality to add to SomeType goes here  
}  
  
extension SomeType: SomeProtocol, AnotherProtocol {  
    // implementation of protocol requirements goes here  
}
```

Capacidades de las extensiones

- Añadir propiedades calculadas (no almacenadas ni observers)
- Definir métodos de instancia y tipo
- Añadir nuevos inicializadores
- Definir subíndices
- Definir y usar nuevos tipos anidados
- Hacer que un tipo existente adopte un protocolo

Añadir propiedades calculadas

```
extension Double {  
    var km: Double { return self * 1_000.0 }  
    var m: Double { return self }  
    var cm: Double { return self / 100.0 }  
    var mm: Double { return self / 1_000.0 }  
    var ft: Double { return self / 3.28084 }  
}
```

```
let oneInch = 25.4.mm  
print("One inch is \(oneInch) meters")  
// Prints "One inch is 0.0254 meters"
```

```
let threeFeet = 3.ft  
print("Three feet is \(threeFeet) meters")  
// Prints "Three feet is 0.914399970739201 meters"
```

Añadir propiedades calculadas

```
let aMarathon = 42.km + 195.m  
print("A marathon is \ (aMarathon) meters long")  
// Prints "A marathon is 42195.0 meters long"
```

Añadir inicializadores

- Se pueden añadir inicializadores de conveniencia
- No se pueden añadir inicializadores designados
- No se pueden añadir desinicializadores

Añadir inicializadores

```
struct Size {  
    var width = 0.0, height = 0.0  
}
```

```
struct Point {  
    var x = 0.0, y = 0.0  
}
```

```
struct Rect {  
    var origin = Point()  
    var size = Size()  
}
```

Añadir inicializadores

```
let defaultRect = Rect()  
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),  
    size: Size(width: 5.0, height: 5.0))
```

Añadir inicializadores

```
extension Rect {  
    init(center: Point, size: Size) {  
        let originX = center.x - (size.width / 2)  
        let originY = center.y - (size.height / 2)  
        self.init(origin: Point(x: originX, y: originY), size: size)  
    }  
}
```

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),  
                      size: Size(width: 3.0, height: 3.0))  
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

Añadir métodos

```
extension Int {  
    func repetitions(task: () -> Void) {  
        for _ in 0..  
            self {  
            task()  
        }  
    }  
}
```

Añadir métodos

```
3.repetitions {  
    print( "Hello!" )  
}  
// Hello!  
// Hello!  
// Hello!
```

Añadir métodos mutantes

```
extension Int {  
    mutating func square() {  
        self = self * self  
    }  
}
```

```
var someInt = 3  
someInt.square()  
// someInt is now 9
```

Añadir subíndices

```
extension Int {  
    subscript(digitIndex: Int) -> Int {  
        var decimalBase = 1  
        for _ in 0..  
            digitIndex {  
            decimalBase *= 10  
        }  
        return (self / decimalBase) % 10  
    }  
}
```

```
746381295[0] // returns 5  
746381295[1] // returns 9  
746381295[2] // returns 2  
746381295[8] // returns 7
```

```
746381295[9]  
// returns 0, as if you had requested:  
0746381295[9]
```

Añadir tipos anidados

```
extension Int {  
  enum Kind {  
    case negative, zero, positive  
  }  
  var kind: Kind {  
    switch self {  
    case 0:  
      return .zero  
    case let x where x > 0:  
      return .positive  
    default:  
      return .negative  
    }  
  }  
}
```


Añadir tipos anidados

```
func printIntegerKinds(_ numbers: [Int]) {  
    for number in numbers {  
        switch number.kind {  
        case .negative:  
            print("-", terminator: "")  
        case .zero:  
            print("0 ", terminator: "")  
        case .positive:  
            print("+ ", terminator: "")  
        }  
    }  
    print("")  
}  
  
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])  
// Prints "+ + - 0 - 0 + "
```

Protocolos

Protocolos

- Permiten definir un listado de métodos, propiedades y otros requisitos que se deben cumplir para garantizar cierta funcionalidad
- No proporcionan la implementación
- Pueden ser adoptados por una clase, estructura o enumeración
- Pueden requerir métodos de instancia o de tipo, propiedades de instancia, operadores y subíndices

Definición de un protocolo

```
protocol SomeProtocol {  
    // protocol definition goes here  
}
```

Adopción de un protocolo

```
struct SomeStructure: FirstProtocol, AnotherProtocol {  
    // structure definition goes here  
}
```

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {  
    // class definition goes here  
}
```

Requerir propiedades

```
protocol SomeProtocol {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
}  
  
protocol AnotherProtocol {  
    static var someTypeProperty: Int { get set }  
}
```

Requerir propiedades

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

```
struct Person: FullyNamed {  
    var fullName: String  
}
```

```
let john = Person(fullName: "John Appleseed")  
// john.fullName is "John Appleseed"
```

Requerir propiedades

```
class Starship: FullyNamed {  
    var prefix: String?  
    var name: String  
    init(name: String, prefix: String? = nil) {  
        self.name = name  
        self.prefix = prefix  
    }  
    var fullName: String {  
        return (prefix != nil ? prefix! + " " : "") + name  
    }  
}
```

```
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")  
// ncc1701.fullName is "USS Enterprise"
```


Requerir métodos

```
protocol RandomNumberGenerator {  
    func random() -> Double  
}
```

```
protocol SomeProtocol {  
    static func someTypeMethod()  
}
```

Requerir métodos

```
class LinearCongruentialGenerator: RandomNumberGenerator {  
    var lastRandom = 42.0  
    let m = 139968.0  
    let a = 3877.0  
    let c = 29573.0  
    func random() -> Double {  
        lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m))  
        return lastRandom / m  
    }  
}
```

Requerir métodos

```
let generator = LinearCongruentialGenerator()  
  
print("Here's a random number: \(generator.random())")  
// Prints "Here's a random number: 0.37464991998171"  
  
print("And another one: \(generator.random())")  
// Prints "And another one: 0.729023776863283"
```

Requerir métodos mutantes

```
protocol Toggleable {  
    mutating func toggle()  
}
```

- Permiten requerir métodos que modifiquen los valores de una instancia en tipos por valor

Requerir métodos mutantes

```
enum OnOffSwitch: Toggable {  
    case off, on  
    mutating func toggle() {  
        switch self {  
            case .off:  
                self = .on  
            case .on:  
                self = .off  
        }  
    }  
}  
  
var lightSwitch = OnOffSwitch.off  
  
lightSwitch.toggle()  
// lightSwitch is now equal to .on
```

Requerir inicializadores

```
protocol SomeProtocol {  
    init(someParameter: Int)  
}
```

Requerir inicializadores

```
class SomeClass: SomeProtocol {  
    required init(someParameter: Int) {  
        // initializer implementation goes here  
    }  
}
```

Requerir inicializadores con herencia

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```


Protocolos como tipos de datos

- Los protocolos generan nuevos tipos de datos
- Se pueden pasar como parámetro a funciones, métodos e inicializadores
- Sirven para definir variables y constantes
- Se pueden usar como tipos de datos en arrays, diccionarios y otras colecciones

Protocolos como tipos de datos

```
class Dice {  
    let sides: Int  
    let generator: RandomNumberGenerator  
    init(sides: Int, generator: RandomNumberGenerator) {  
        self.sides = sides  
        self.generator = generator  
    }  
    func roll() -> Int {  
        return Int(generator.random() * Double(sides)) + 1  
    }  
}
```

Protocolos como tipos de datos

```
var d6 = Dice(sides: 6, generator:
LinearCongruentialGenerator())

for _ in 1...5 {
    print("Random dice roll is \((d6.roll())")
}

// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

Delegación

- Es un patrón de diseño que permite que una clase o estructura delegue algunas de sus responsabilidades en otro tipo de dato
- Se implementa definiendo un protocolo que encapsule las responsabilidades
- Como adopta el protocolo, el delegado garantiza que provee la funcionalidad necesaria

Genéricos

Tipos de datos genéricos

- Permiten escribir código que admita cualquier tipo de dato
- Gran parte de la librería estándar de Swift está escrita mediante genéricos (los tipos `Array` o `Dictionary` son genéricos, por ejemplo)

Necesidad de los genéricos

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Necesidad de los genéricos

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \({someInt}), and anotherInt is now \({anotherInt}")
// Prints "someInt is now 107, and anotherInt is now 3"
```


Necesidad de los genéricos

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Funciones genéricas

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Funciones genéricas

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

Funciones genéricas

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

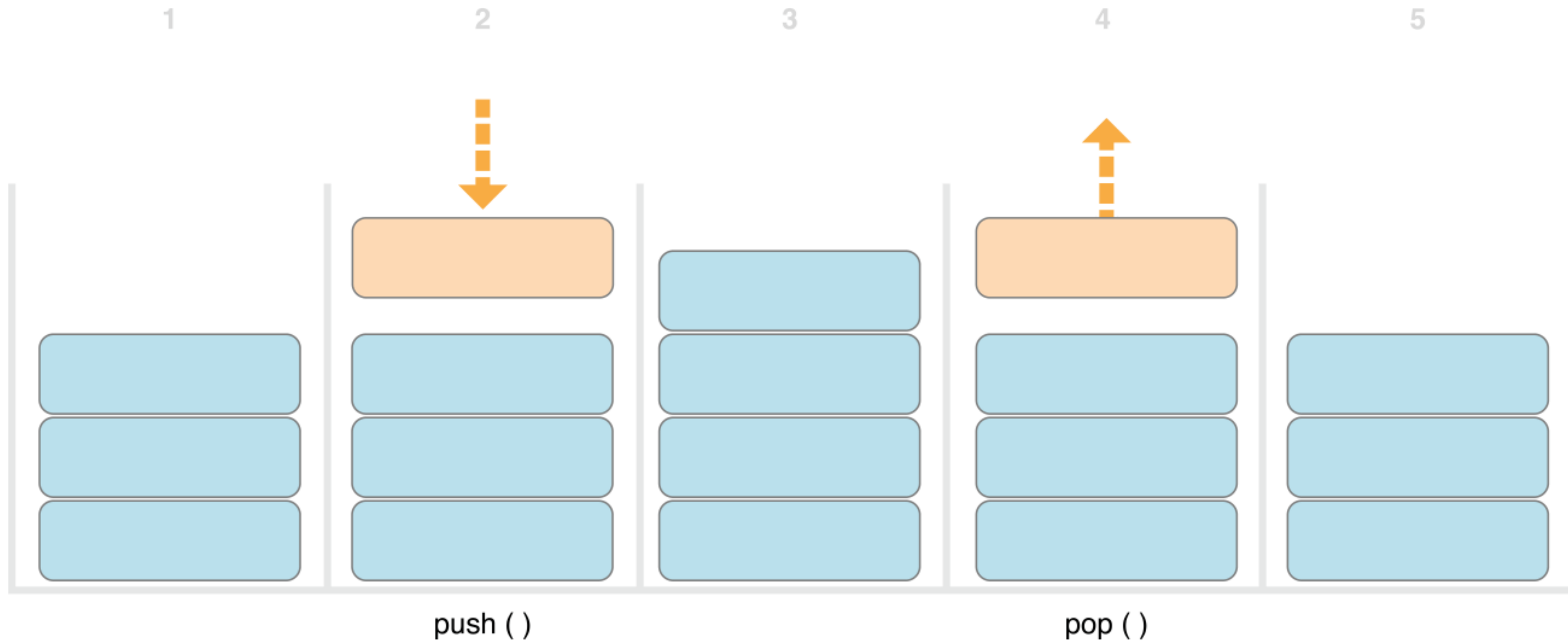
Parámetros de tipo

- Son los que se añaden entre `<>` a la definición de las funciones genéricas, por ejemplo `<T>` de `swapTwoValues(_:_:)`
- Se sustituyen por el tipo concreto al llamar a la función
- Podemos poner más de uno, separados por comas
- Pueden tener cualquier nombre y ser descriptivos (como en `Dictionary<Key, Value>` o `Array<Element>`) o simples letras (T, U, V, ...)

Tipos genéricos

- Igual que podemos parametrizar una función, podemos hacer lo mismo con una enumeración, estructura o clase

Pila



Tipos de datos genéricos

```
struct IntStack {  
    var items = [Int]()  
    mutating func push(_ item: Int) {  
        items.append(item)  
    }  
    mutating func pop() -> Int {  
        return items.removeLast()  
    }  
}
```

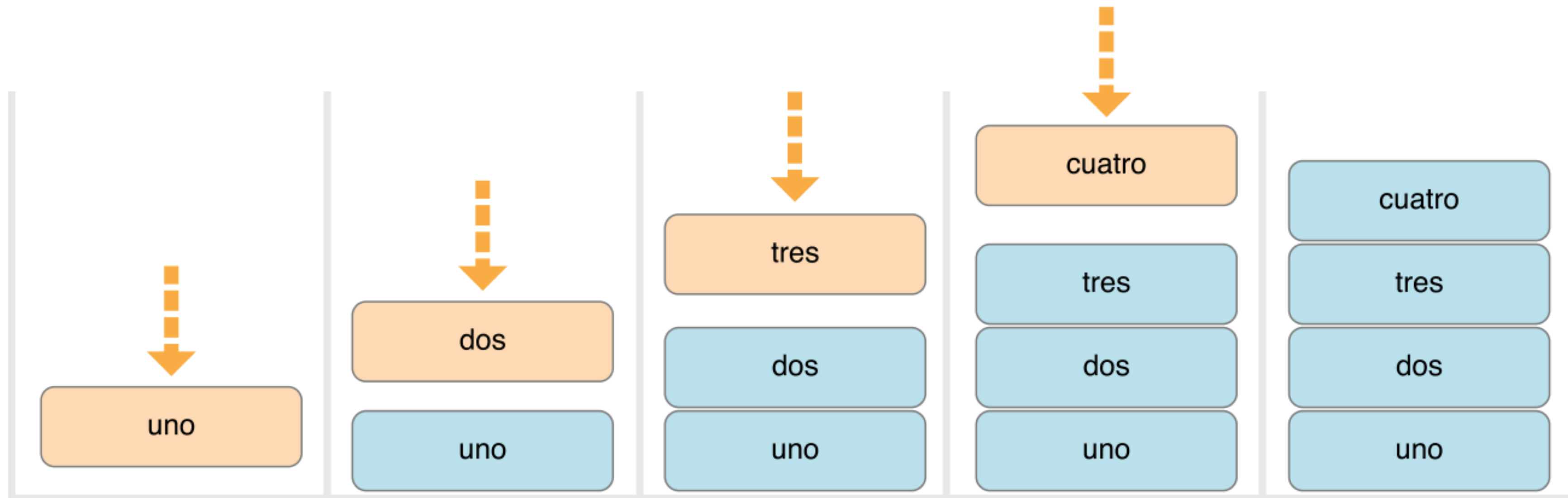

Tipos de datos genéricos

```
struct Stack<Element> {  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
}
```

Tipos de datos genéricos

```
var stackOfStrings = Stack<String>()  
stackOfStrings.push("uno")  
stackOfStrings.push("dos")  
stackOfStrings.push("tres")  
stackOfStrings.push("cuatro")  
// the stack now contains 4 strings
```

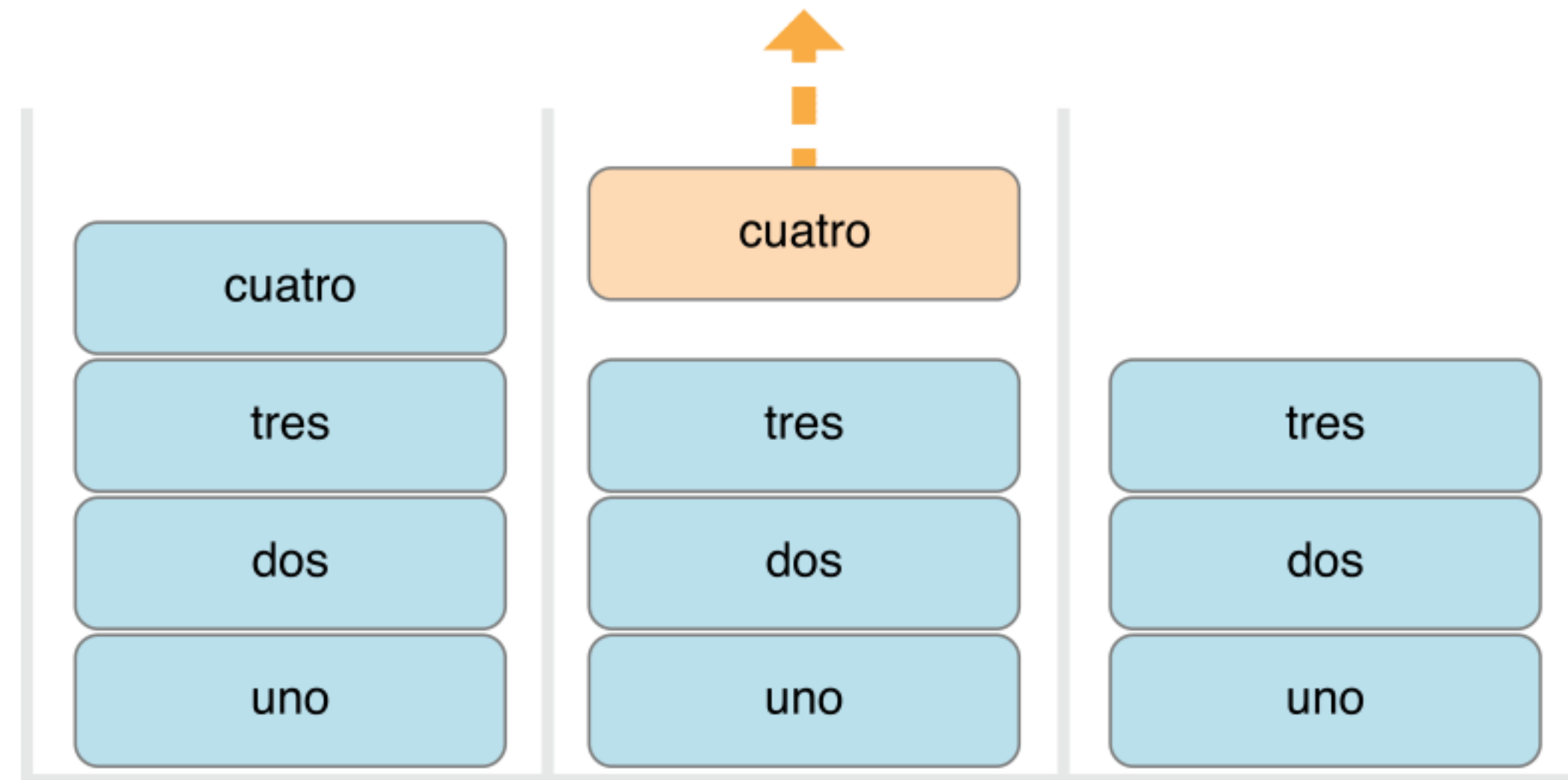
Tipos de datos genéricos



Tipos de datos genéricos

```
let fromTheTop = stackOfStrings.pop()  
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

Tipos de datos genéricos



Extendiendo tipos de datos genéricos

- Al extender un tipo genérico, no necesitamos añadir la parametrización, ya tendremos disponible los parámetros del tipo original

Extendiendo tipos de datos genéricos

```
extension Stack {  
    var topItem: Element? {  
        return items.isEmpty ? nil : items[items.count - 1]  
    }  
}  
  
if let topItem = stackOfStrings.topItem {  
    print("The top item on the stack is \(topItem).")  
}  
// Prints "The top item on the stack is tres."
```

Restricciones de tipo

- Permiten exigir que los parámetros de un genérico cumplan ciertas reglas
- Podemos exigir que hereden de cierta clase o que adopten cierto protocolo

Restricciones de tipo

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {  
    // function body goes here  
}
```

Restricciones de tipo

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

Restricciones de tipo

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
// Prints "The index of llama is 2"
```

Restricciones de tipo

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind { // error, == no está implementado para T  
            return index  
        }  
    }  
    return nil  
}
```

Protocolo Equatable

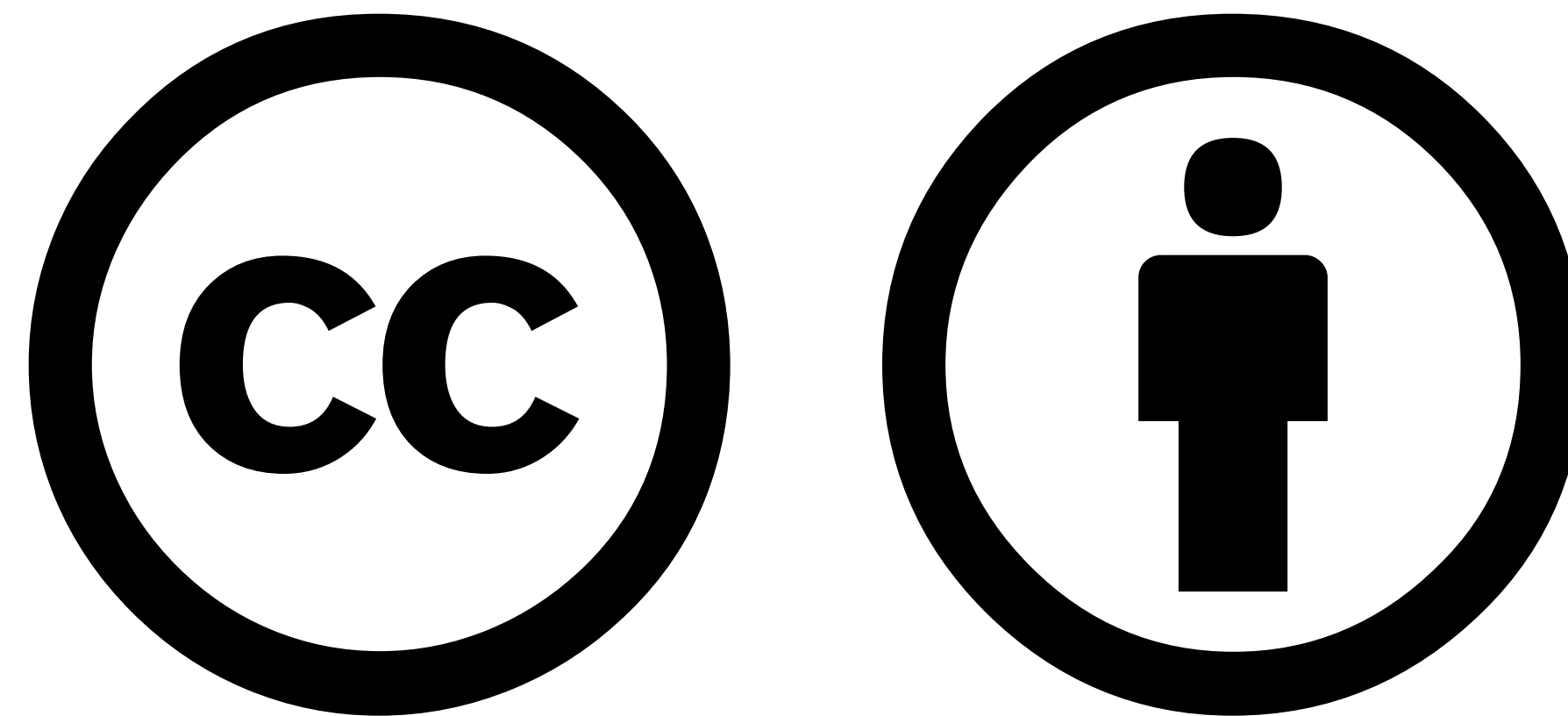
- La librería estándar de Swift define el protocolo `Equatable`, que exige a cualquier tipo que lo adopte a implementar los operadores igual que (`==`) y distinto de (`!=`) para comparar entre sí dos valores de dicho tipo de dato
- Todos los tipos estándar de Swift soportan este protocolo

Restricciones de tipo

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

Restricciones de tipo

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])  
// doubleIndex is an optional Int with no value, because 9.3 isn't in the array  
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])  
// stringIndex is an optional Int containing a value of 2
```



Excepto si se especifica lo contrario, esta presentación está bajo licencia

<https://creativecommons.org/licenses/by/4.0/>

© 2017 Ion Jaureguialzo Sarasola. Algunos derechos reservados.