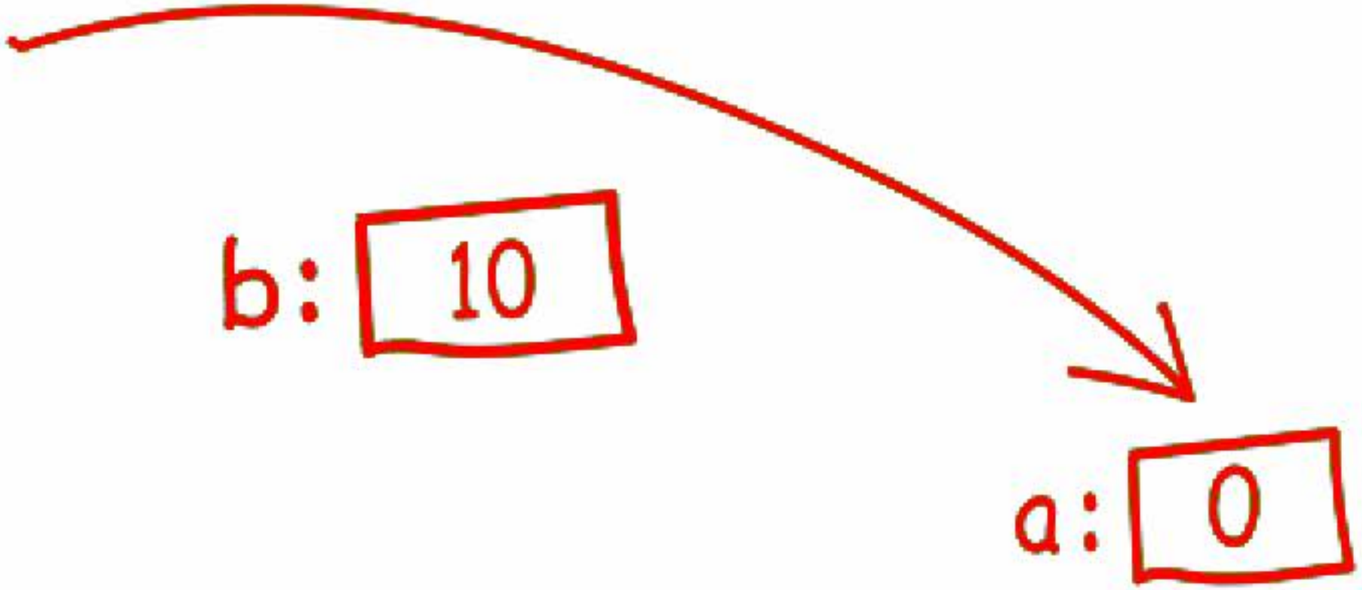
Struct, Class, Tuples, and Enum

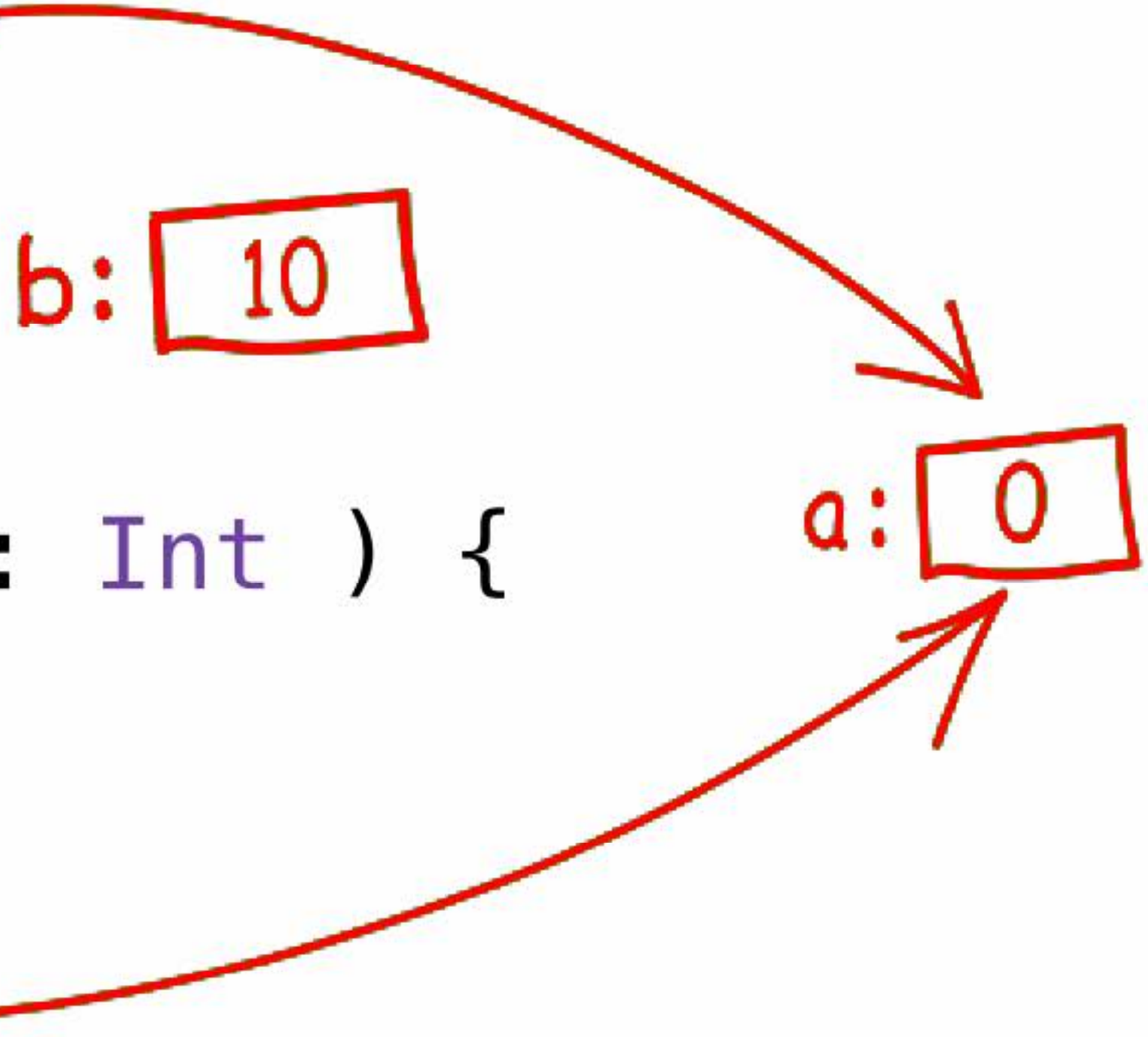
Allen Holub

http://holub.com | Allen Holub | @allenholub

```
var a = 0
var b = a
b = 10
```



```
var a = 0
var b = a
b = 10
func f( var a: Int ) {
    a = -1
}
f(a)
```



```
var a = 0
var b = a
b = 10
func f( var a: Int ) {
    a = -1
}
f(a)
```

b: 10

a: 0

a: 0

a==0

```
var a = 0
var b = a
b = 10
func f( var a: Int ) {
    a = -1
}
f(a)
```

b: 10

a: 0

a: -1

a==0

```
var a = 0
var b = a
b = 10
func f( var a: Int ) {
    a = -1
}
f(a)
```

b: 10

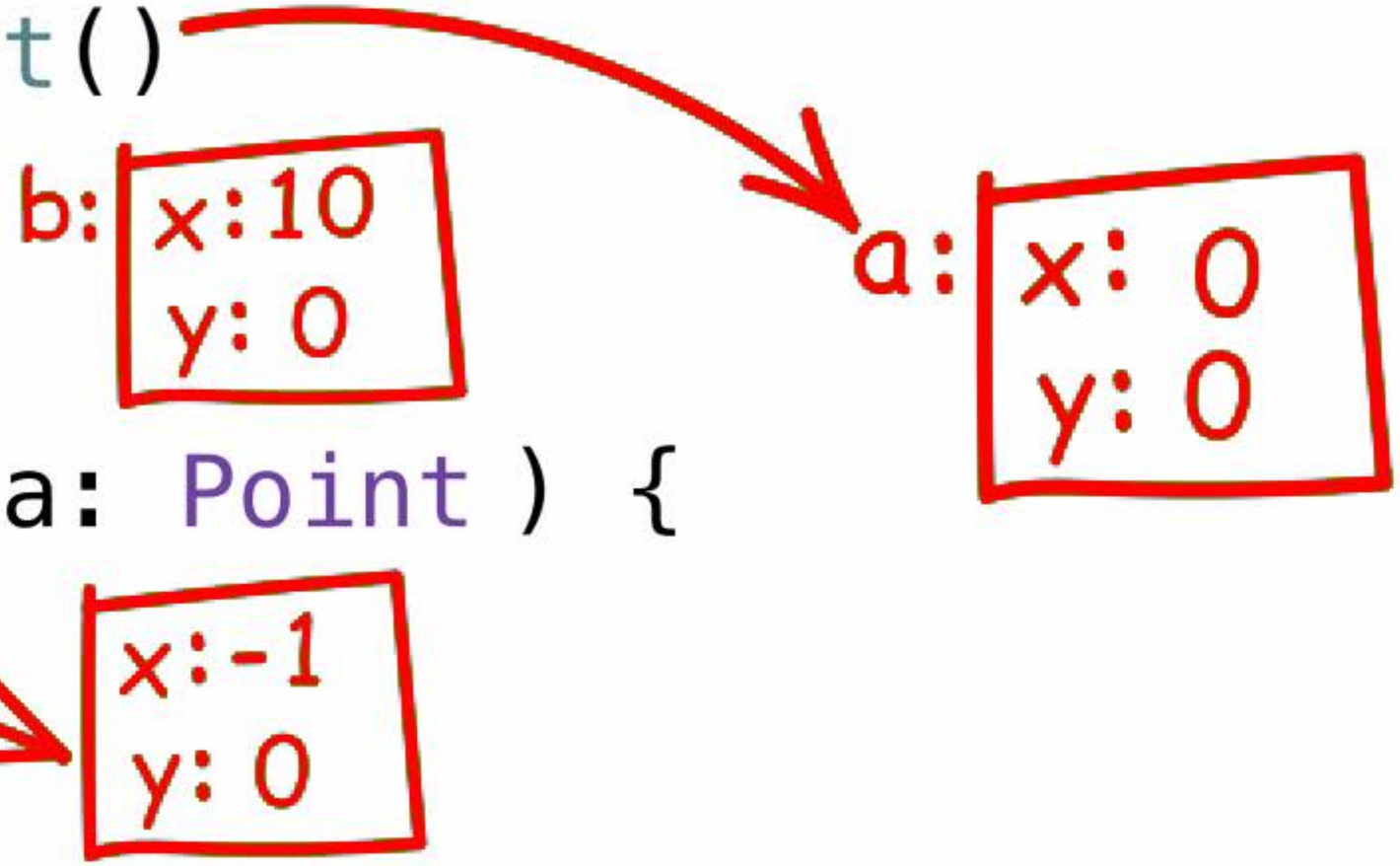
a: 0

a: -1

a == 0

a == 0

```
struct Point { var x=0, y=0 }
var a = Point()
var b = a
b.x = 10
func f( var a: Point ) {
    a.x = -1
}
f(a)
```



```
struct Point { var x=0, y=0 }
var a = Point()
var b = a
b.x = 10
func f( var a: Point ) {
    a.x = -1
}
f(a)
```

a.x == 0

a.x == 0

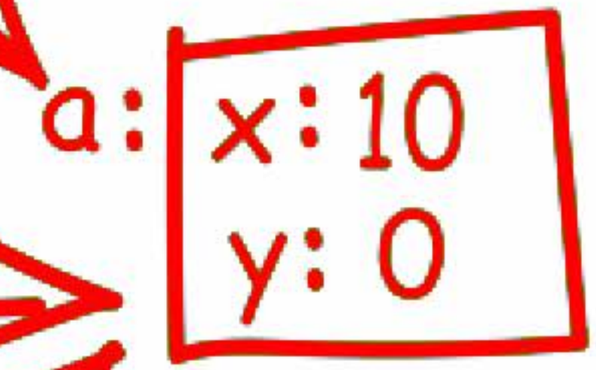
b: x:10 y: 0

a: x: 0 y: 0

x:-1 y: 0

```
class Point { var x=0, y=0 }
var a = Point()
var b = a
b.x = 10
func f( var a: Point ) {
    a.x = -1
}
f(a)
```

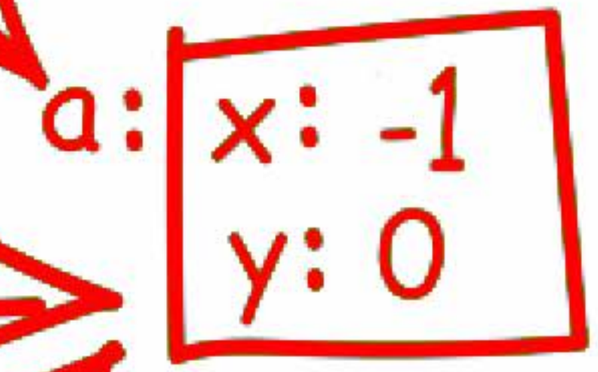
a.x == 10



```
class Point { var x=0, y=0 }
var a = Point()
var b = a
b.x = 10
func f( var a: Point ) {
    a.x = -1
}
f(a)
```

a.x == 10

a.x == -1



```
class Circle {
    var center = Point()
    var radius = 0.0
    init(c:Point,r:Point){ center = c; radius = r }
}
```

```
let smallest = Circle()

let larger = Circle( c:Point(), r:5.0 )
```

```
class Circle {
   var center = Point()
   var radius = 0.0
   init(c:Point,r:Point){ center = c; radius = r }
   deinit { /*…*/ }
   final func moveTo (center:Point) {
      self.center = center
   }


   lazy     var  offset =  getDefault()
   func offset(here:Point){ offset = here }
}
```

```
class Circle {
   var center = Point()
   var radius = 0.0
   init(c:Point,r:Point){ center = c; radius = r }
   deinit { /*…*/ }
   final func moveTo (center:Point) {
      self.center = center
   }


   static  var  offset =  getDefault()
   class   func offset(here:Point){ offset = here }
}
```

```
class Circle {
   var center = Point()
   var radius = 0.0
   init(c:Point,r:Point){ center = c; radius = r }
   deinit { /*…*/ }
   final func moveTo (center:Point) {
      self.center = center
   }


   static  var  offset =  getDefault()
   static  func offset(here:Point){ offset = here }
}
```

```
struct  Circle {
   var center = Point()
   var radius = 0.0
   init(c:Point,r:Point){ center = c; radius = r }


   mutating  func moveTo (center:Point) {
      self.center = center
   }
   mutating func reset() { self = Circle() }
   static  var  offset =  getDefault()
   static  func offset(here:Point){ offset = here }
   struct Nested { /*…*/ }
}
```

```
struct  Circle {
   var center = Point()
   var radius = 0.0
   init(c:Point,r:Point){ center = c; radius = r }


   mutating  func moveTo (center:Point) {
      self.center = center
   }
   mutating func reset() { self = Circle() }
   static  var  offset =  getDefault()
   static  func offset(here:Point){ offset = here }
   enum  Nested { /*…*/ }
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next

{ /*…*/
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next?.next.i

{ /*…*/
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next?.optDict

{ /*…*/
}
```

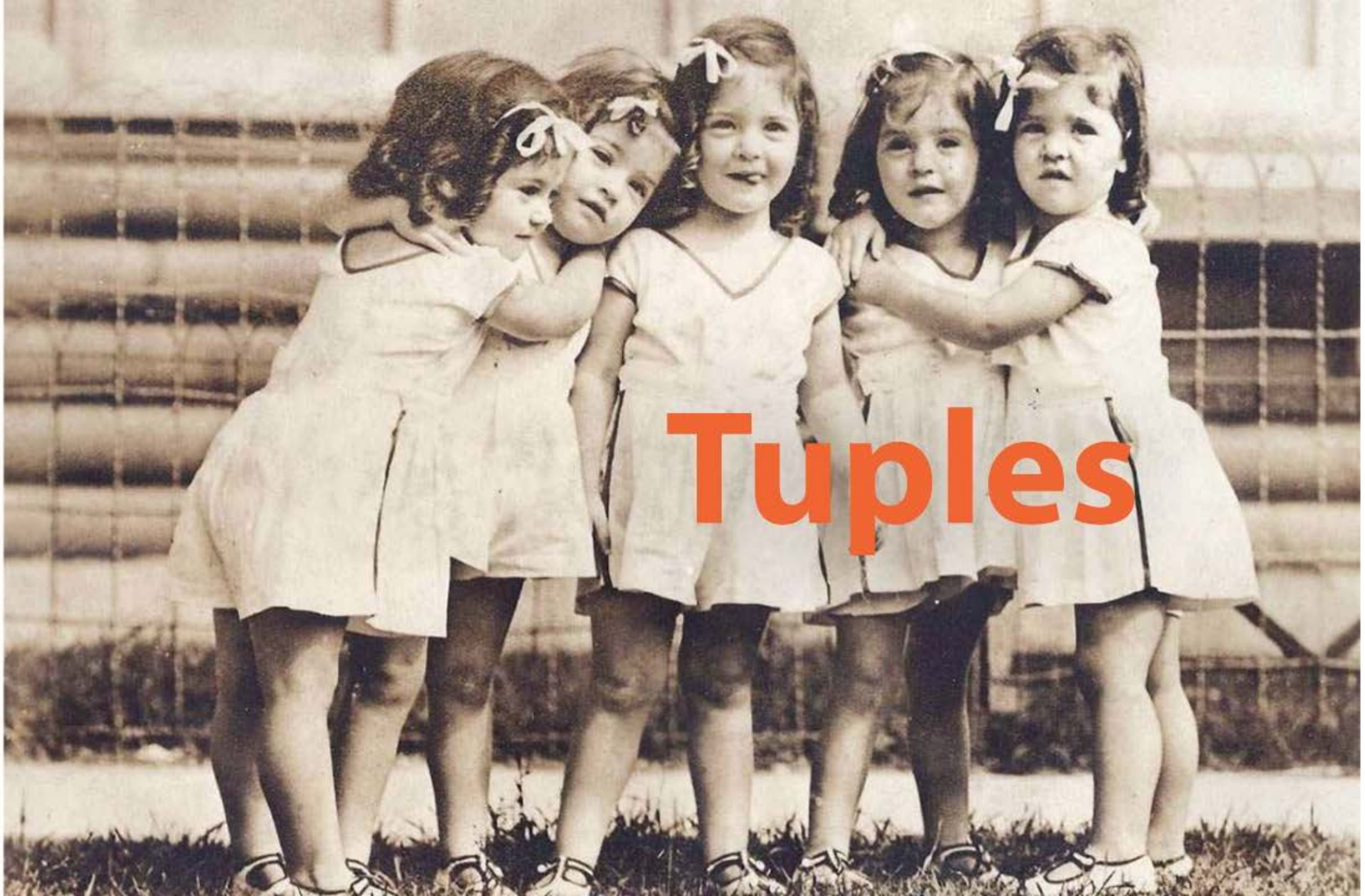
```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next?.optDict ?.count

{ /*…*/
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next?.optDict ?["k"]

{ /*…*/
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
if let a =
     obj.next?.optDict ?["k"]?.isEmpty

{ /*…*/
}
```

```
class Node {
    var i = 9
    var optDict: [String:String]?
    var next: Node?
    func successor()->Node?{return next}
}
let obj = Node()
let optObj = obj.next

optObj!.successor()
optObj?.successor()
```

```
class HttpStatus {
    var description: String
    var code : Int
    init( _ code:Int, _ description:String){
        self.code = code
        self.description = description;
    }
}

let statii = [ HttpStatus(404, "Not Found"),
               HttpStatus(200, "Okay")
             ]
```



Tuples

```
let http404Error = ( 404, "Not Found")
```

```
http404Error.0
http404Error.1
```

```
let (status, desc )= http404Error
print("\(status): \(desc)")
```

```swift
let http404Error = ( 404, "Not Found")
```

```swift
http404Error.0
http404Error.1
```

```swift
let (status, _ )= http404Error
print("\(status)")
```

```
let http404Error =
  (code: 404, description:"Not Found")


 http404Error.code
 http404Error.description


 let (status, _ )= http404Error
 print("\(status)")
```

```
func fetch()-> (code:Int,desc:String) {
    return (200, "Okay")
}

let status = fetch()
status.code
status.desc

let (error,description) = fetch()
print("\(error): \(description)")
```

```
enum CompassPoint {
    case North
    case South
    case East, West
}
```

:CompassPoint

```
var direction = CompassPoint.North
direction = .East
```

```
enum CompassPoint {
    case North
    case South
    case East, West
}

var direction = CompassPoint.North
direction = .East

switch direction {
case .East:  print("East")
default:     print("Not East")
}
```

```
enum PostalCode {
    case US(Int,Int)
    case UK(String)
    case CA(code:String)
}

var somewhere =  PostalCode.US(94707,2625)
```

```
enum PostalCode {
    case US(Int,Int)
    case UK(String)
    case CA(code:String)
}

var somewhere = PostalCode.UK("SW1A 1AA")
```

```
enum PostalCode {
    case US(Int,Int)
    case UK(String)
    case CA(code:String)
}

var somewhere =  PostalCode.CA(code:"V5K 0A1")
```

```
enum PostalCode {
    case US(Int,Int)
    case UK(String)
    case CA(code:String)
}

var somewhere =  PostalCode.CA(code:"V5K 0A1")

switch somewhere {
  case .UK (let s): print("\(s)")
  case .US (let loc, var route ):
      print("\(loc)-\(route)")
  case .CA: break;
}
```

```
enum PostalCode {
    case US(Int,Int)
    case UK(String)
    case CA(code:String)
}

var somewhere =  PostalCode.CA(code:"V5K 0A1")

switch somewhere {
  case .UK (let s): print("\(s)")
  case .US (let loc, _ ):
      print("\(loc)")
  case .CA: break;
}
```

```
enum ASCIIControls : Character {
    case Newline  = "\n"
    case Carriage = "\r"
    case Tab      = "\t"
}
```

```
enum Planet: Int {
    case Mercury = 1,
         Venus, Earth, Mars, Jupiter,
         Saturn, Uranus, Neptune
}

let x = Planet.Earth.rawValue

if let aPlanet = Planet(rawValue: 9) {
    print("\(aPlanet.rawValue) exists")
}
```

```
enum Dimension {
    case DISTANCE( Int )
    init( distance: Int ){self = DISTANCE(height + 100)}
    func value() -> Int {
        switch self {
        case .DISTANCE ( let value ): return value
        }
    }
}

let aDistance = Dimension.DISTANCE(10)
aDistance.value()        // 10

let anotherDistance = Dimension(distance: 10)
anotherDistance.value()  // 110!
```

```swift
enum ConnectionState {
    case closed, opening, open, closing
    mutating func next() {
        switch self {
        case closed:  self = opening
        case opening: self = open
        case open:    self = closing
        case closing: self = closed
        }
    }
}

var state = ConnectionState.closed
state.next()    // opening
state.next()    // open
state.next()    // closing
state.next()    // closed
```