Interactive Graphics

CSCI B581 – Spring 2018

Lab 07 – more practice with Swift

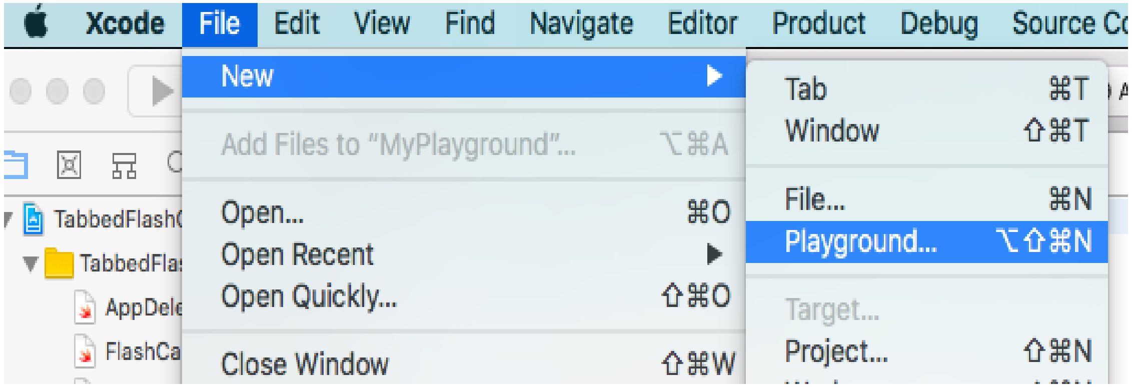
Instructor:
Mitja Hmeljak,
http://pages.iu.edu/~mitja
mitja@indiana.edu

let's try a Swift Playground

In Xcode,

select the menu

File → New → File:



and select

iOS →

Source →

Playground

then click Next...

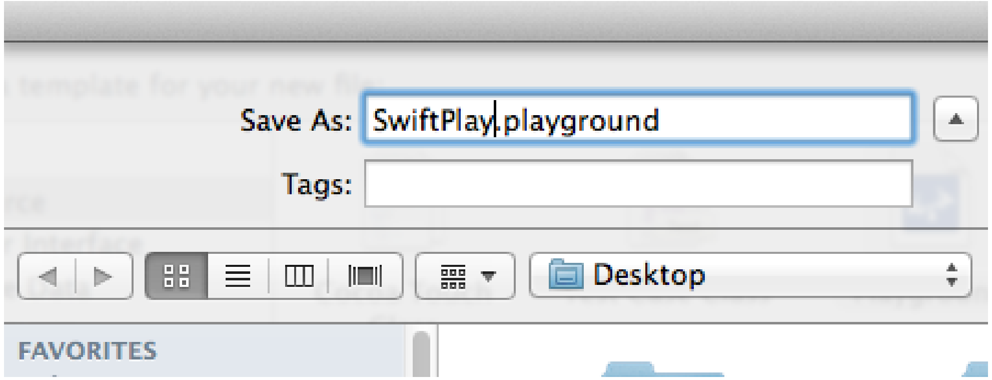
a Swift Playground

name it

" lab07username.playground ",

(or something similar)

and save to Desktop:



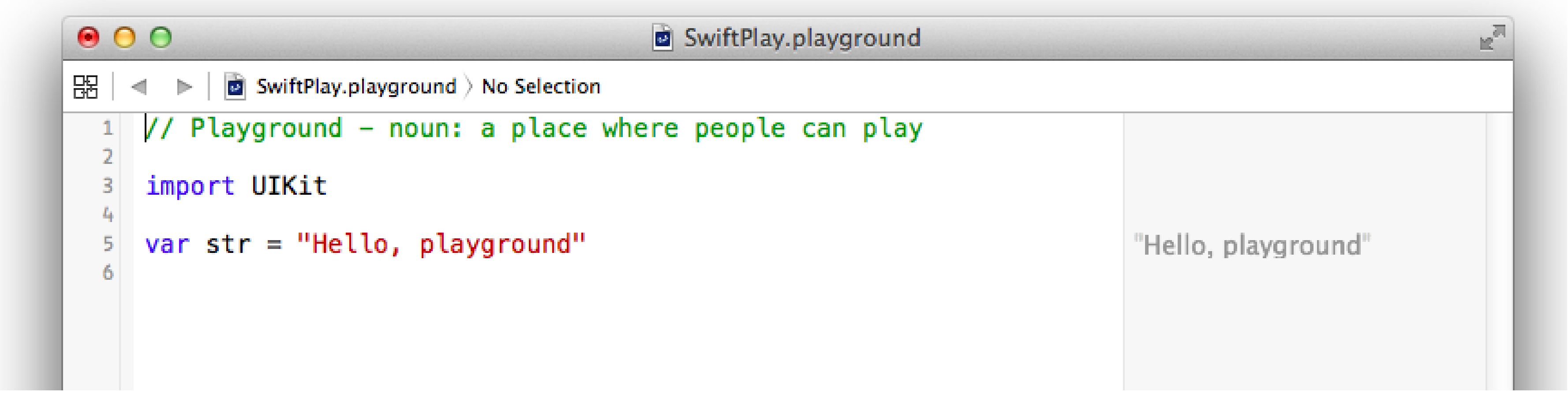
Save As: SwiftPlay.playground

Tags:

Desktop

a Playground in Xcode

what you type is what you get →
on the right side of the window



```
// Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

"Hello, playground"

Swift – the language

compared to Python and
compared to C-derived languages (Java, C++, etc.)

in Swift – just like in Python, and *unlike* C-derived languages:

- there is no *main()* function
- there is no need for *;* (semicolons) to terminate a line

in Swift – just like in C-derived languages – and *unlike* Python:

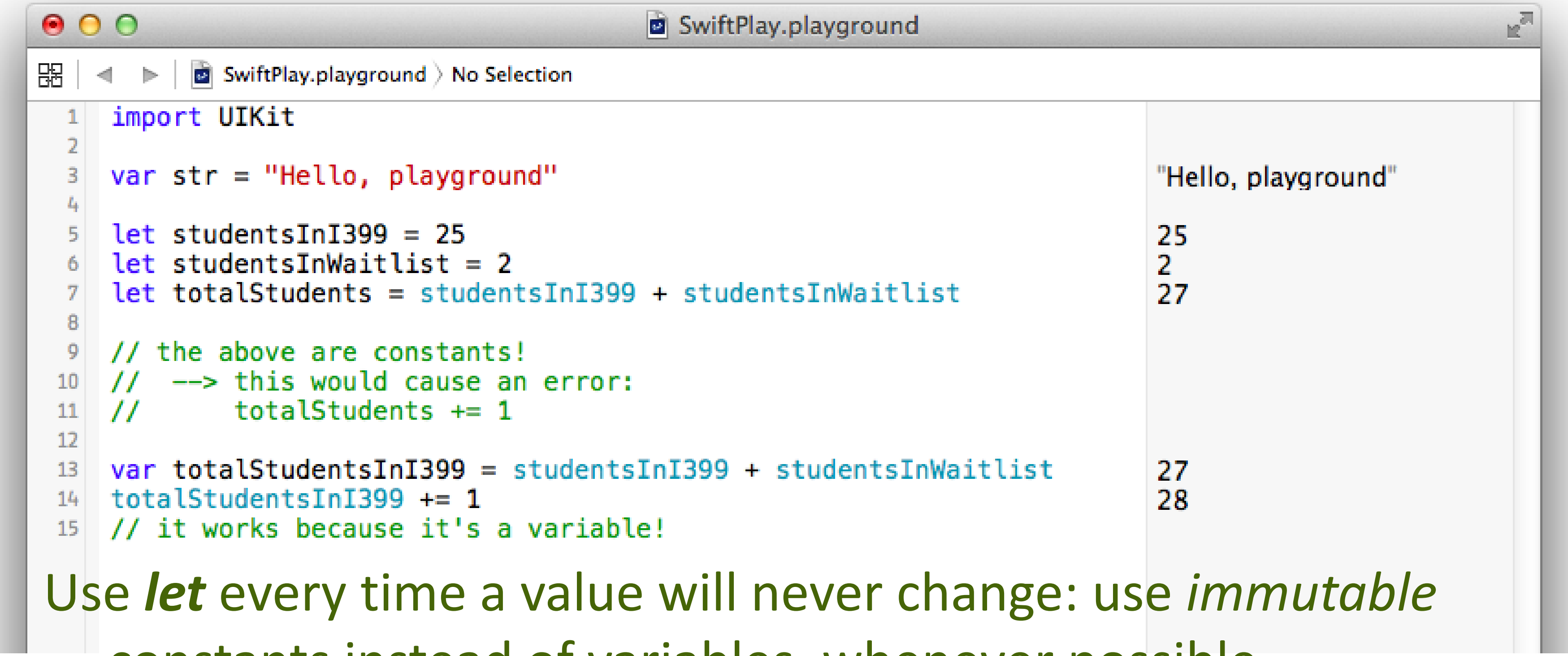
- there is no compulsory indentation (but it helps visual clarity)
- blocks don't depend on indentation level (safer!)

in Swift – just like in C-derived languages – and *unlike* Python:

- *{curly brackets}* are *required* to delimit code blocks.
  For example in *if* statements.

Swift constants and variables

declarations using ***let*** (constants) and ***var*** (variables) keywords:



```
import UIKit

var str = "Hello, playground"                                           "Hello, playground"

let studentsInI399 = 25                                                 25
let studentsInWaitlist = 2                                              2
let totalStudents = studentsInI399 + studentsInWaitlist                 27

// the above are constants!
//   --> this would cause an error:
//       totalStudents += 1

var totalStudentsInI399 = studentsInI399 + studentsInWaitlist           27
totalStudentsInI399 += 1                                                28
// it works because it's a variable!
```

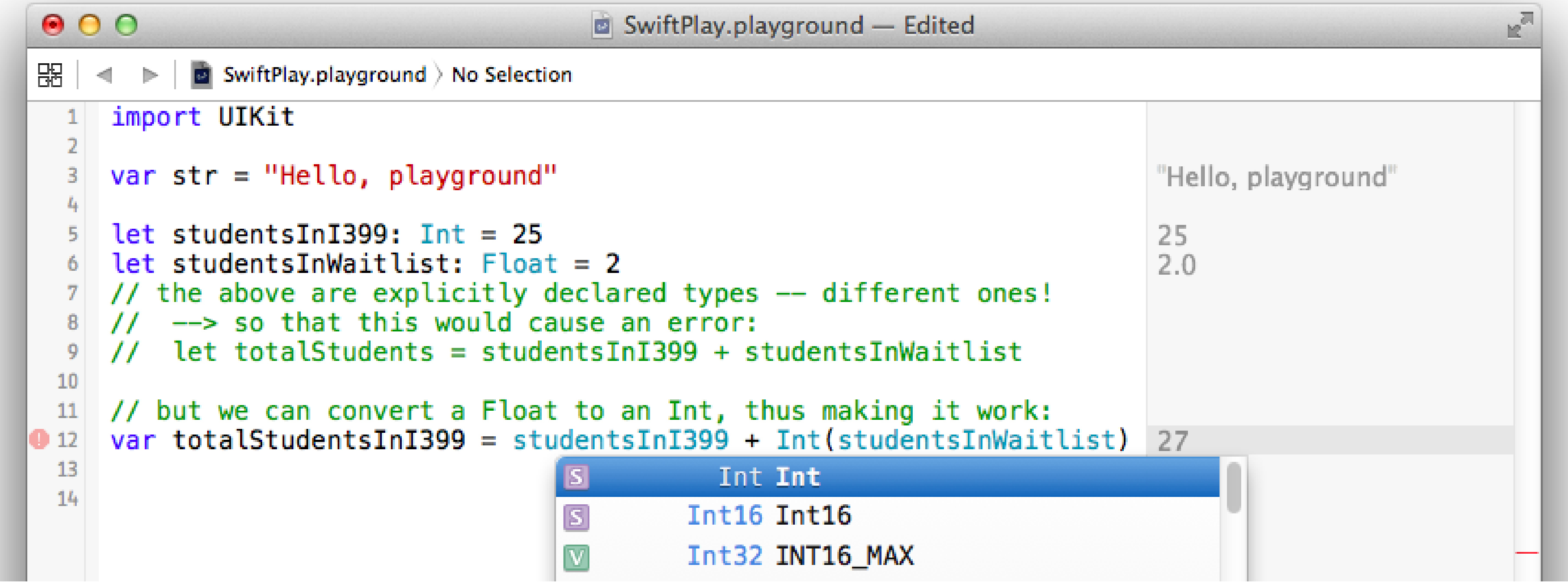
Use ***let*** every time a value will never change: use *immutable* constants instead of variables, whenever possible.

Makes code **safer** (e.g. in a multithreaded environment) and **cleaner** (i.e. more readable, knowing a value won't change).

Swift typing for constants and variables

the type of a Swift variable (or constant) can be either *explicit* as in *C*, or *inferred* as in *Python* (as seen in the previous slide).

floating-point numbers can be converted to integers, and *Int()* always *truncates* the numbers: it does not round.



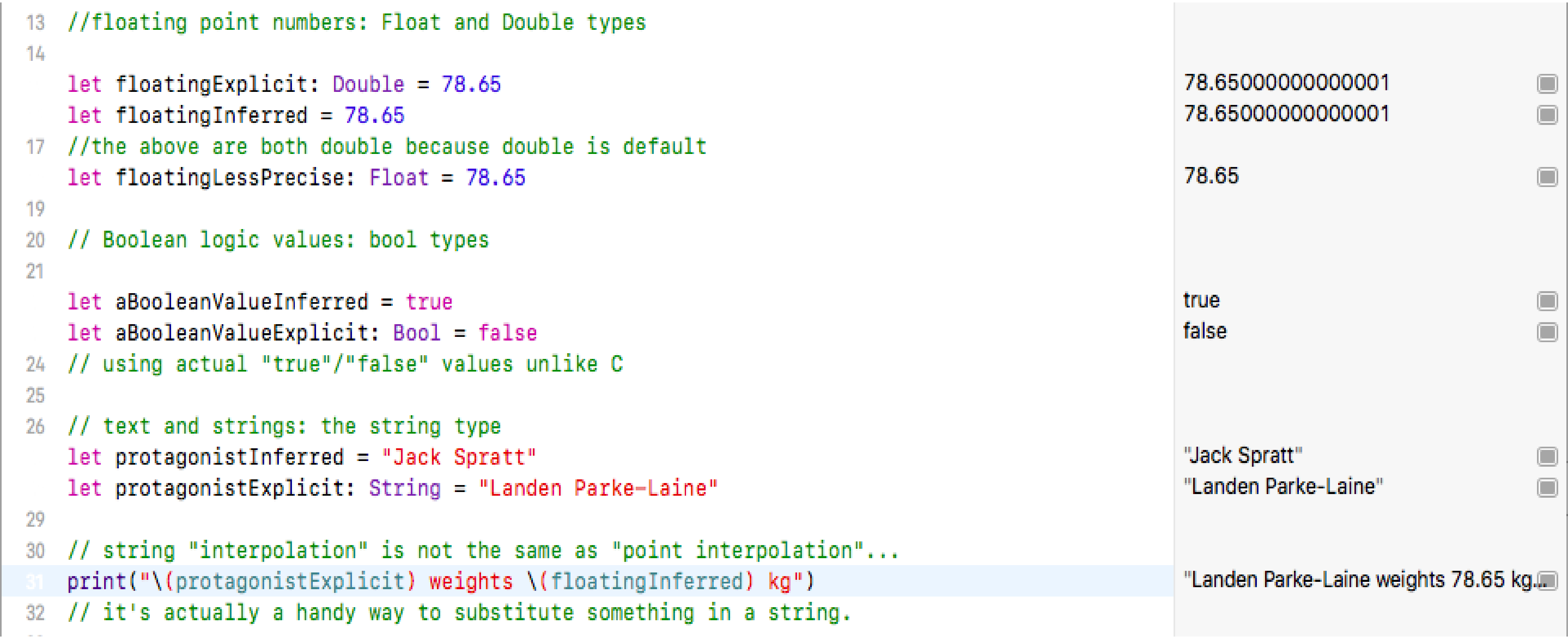
SwiftPlay.playground — Edited
SwiftPlay.playground 〉 No Selection

```
import UIKit

var str = "Hello, playground"

let studentsInI399: Int = 25
let studentsInWaitlist: Float = 2
// the above are explicitly declared types -- different ones!
//  --> so that this would cause an error:
//  let totalStudents = studentsInI399 + studentsInWaitlist

// but we can convert a Float to an Int, thus making it work:
var totalStudentsInI399 = studentsInI399 + Int(studentsInWaitlist)
```

```
"Hello, playground"

25
2.0

27
```

basic Swift types

floats, doubles, bools, strings:



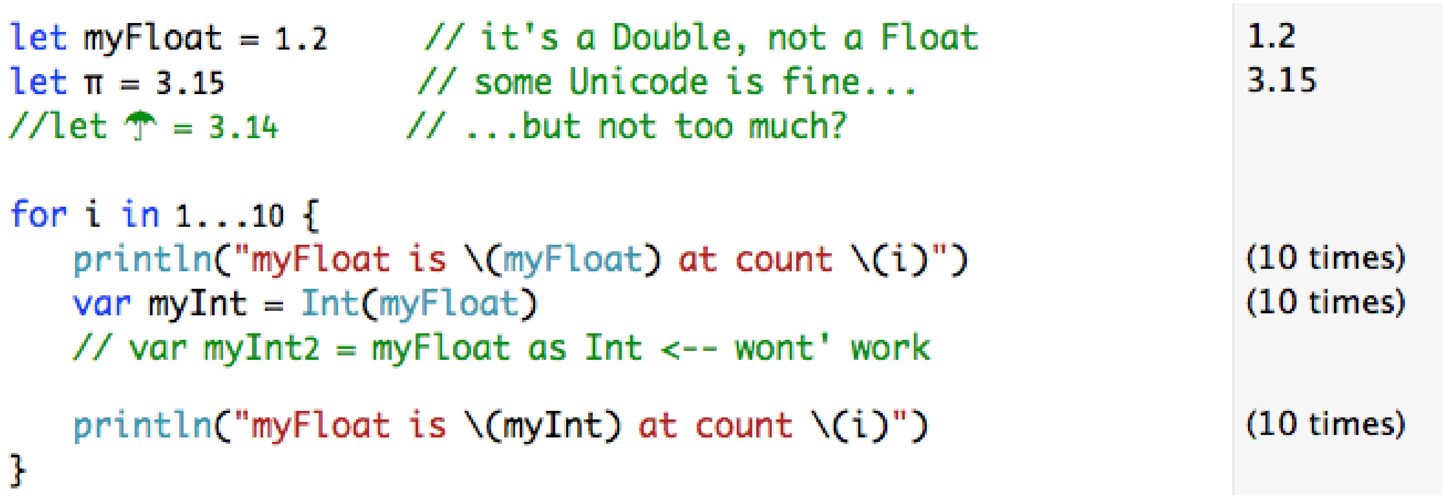
```
//floating point numbers: Float and Double types

let floatingExplicit: Double = 78.65
let floatingInferred = 78.65
//the above are both double because double is default
let floatingLessPrecise: Float = 78.65

// Boolean logic values: bool types

let aBooleanValueInferred = true
let aBooleanValueExplicit: Bool = false
// using actual "true"/"false" values unlike C

// text and strings: the string type
let protagonistInferred = "Jack Spratt"
let protagonistExplicit: String = "Landen Parke-Laine"

// string "interpolation" is not the same as "point interpolation"...
print("\(protagonistExplicit) weights \(floatingInferred) kg")
// it's actually a handy way to substitute something in a string.
```

```
78.65000000000001
78.65000000000001

78.65


true
false


"Jack Spratt"
"Landen Parke-Laine"


"Landen Parke-Laine weights 78.65 kg...
```

Swift variable type declaration

the ***type*** of a Swift variable or constant can be either:

explicit (as in *C*), or ***inferred*** as in *Python*



```
let myFloat = 1.2      // it's a Double, not a Float         1.2
let π = 3.15           // some Unicode is fine...            3.15
//let ☂ = 3.14         // ...but not too much?

for i in 1...10 {
    println("myFloat is \(myFloat) at count \(i)")           (10 times)
    var myInt = Int(myFloat)                                 (10 times)
    // var myInt2 = myFloat as Int <-- wont' work

    println("myFloat is \(myInt) at count \(i)")             (10 times)
}
```

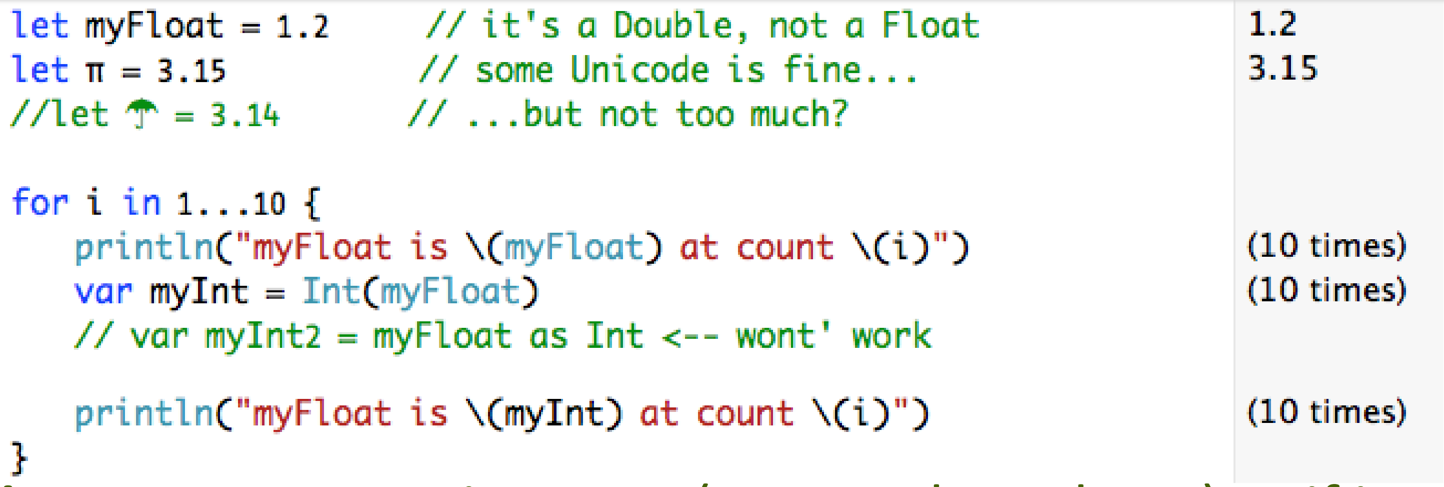
casting vs. **conversion**:

floating-point numbers can be **converted** to integers

using the type name to initialize another variable e.g. *Int(33.22)*

Int() always *truncates* the numbers: it does *not* round.

Swift type casting



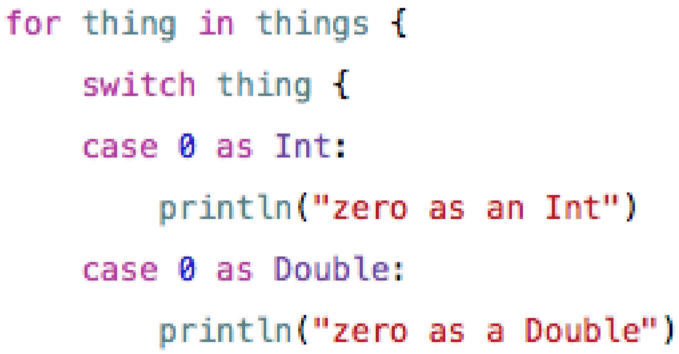
```
let myFloat = 1.2      // it's a Double, not a Float              1.2
let π = 3.15           // some Unicode is fine...                 3.15
//let ☂ = 3.14         // ...but not too much?

for i in 1...10 {
    println("myFloat is \(myFloat) at count \(i)")               (10 times)
    var myInt = Int(myFloat)                                      (10 times)
    // var myInt2 = myFloat as Int <-- wont' work

    println("myFloat is \(myInt) at count \(i)")                 (10 times)
}
```

casting == to treat an instance (e.g. myFloat above) as if it were a different *superclass* or *subclass* from somewhere else in *its own class hierarchy*.

casting wouldn't work in the above

example: *"myFloat as Int"* gives an error.

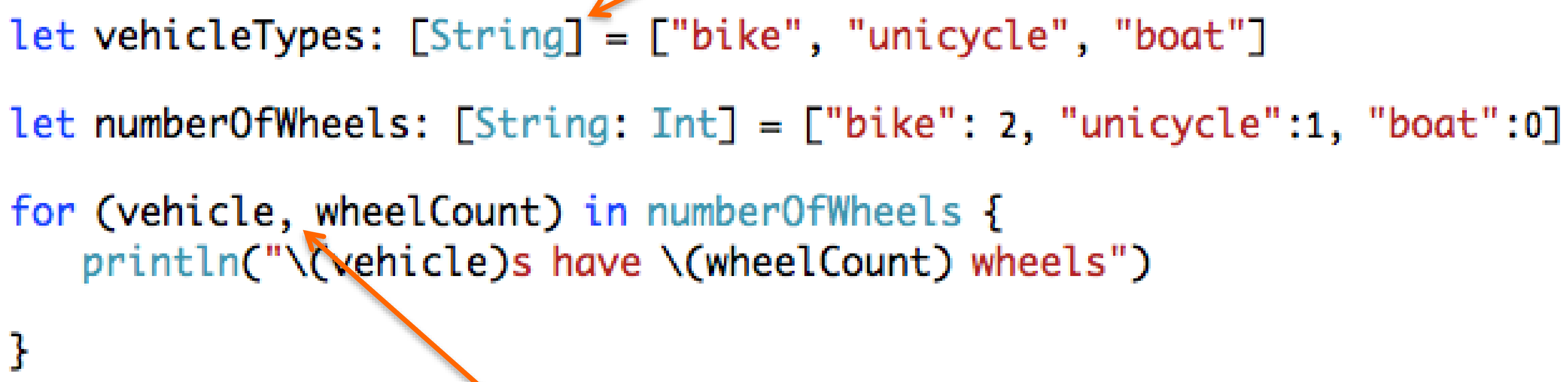
An example of where casting does work:



```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
```

Swift collection types

arrays and **dictionaries** are *typed* collections



```
let vehicleTypes: [String] = ["bike", "unicycle", "boat"]

let numberOfWheels: [String: Int] = ["bike": 2, "unicycle":1, "boat":0]

for (vehicle, wheelCount) in numberOfWheels {
    println("\(vehicle)s have \(wheelCount) wheels")

}
```

tuples are groupings of values

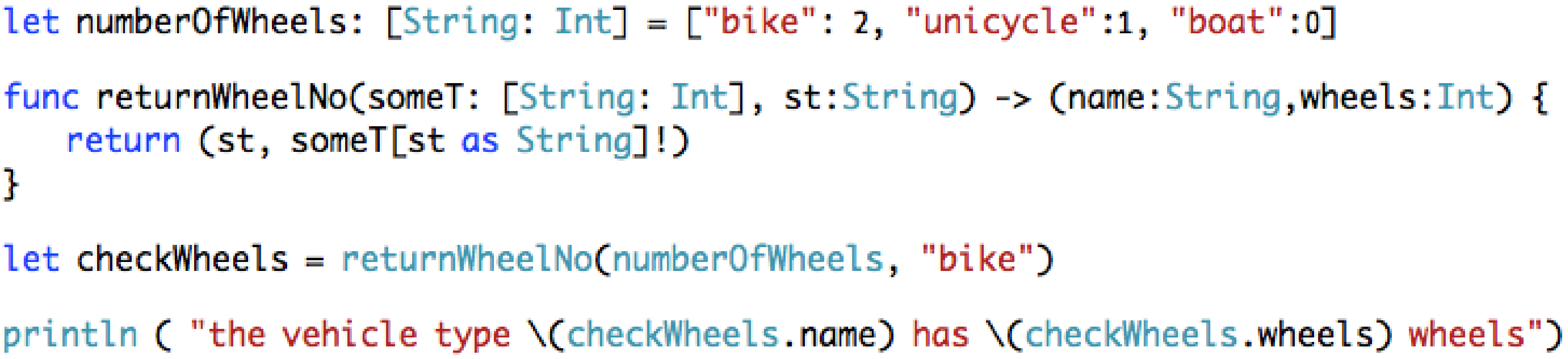
... they are not a separate data type.

Tuples are useful to pass multiple values around.

Swift tuples

named tuples are groupings of values

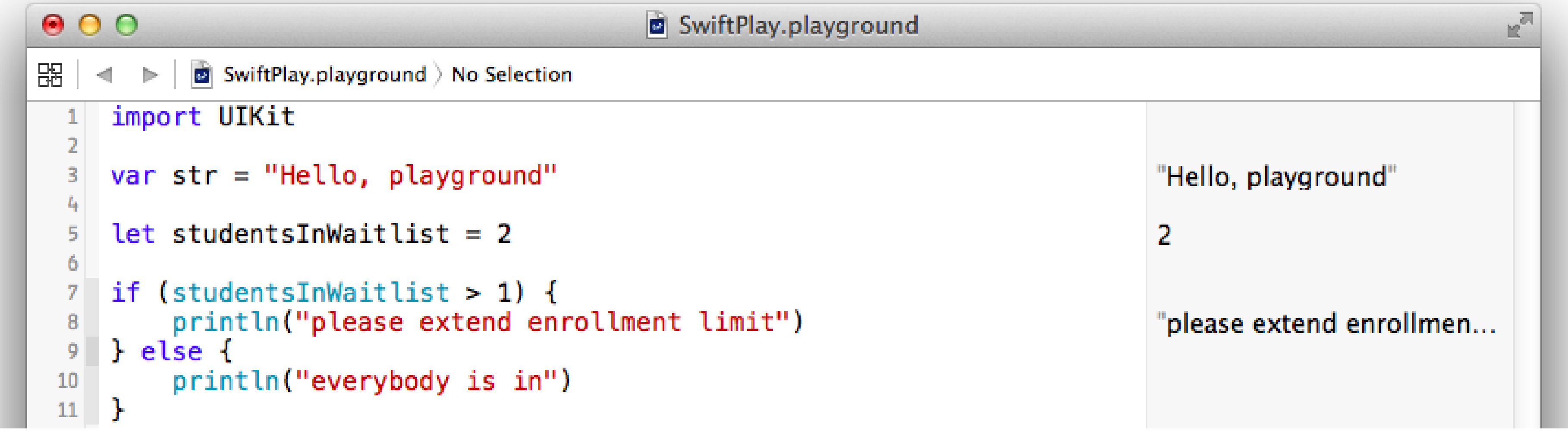
... that can be used for example to return multiple values and refer to them separately:



```
let numberOfWheels: [String: Int] = ["bike": 2, "unicycle":1, "boat":0]

func returnWheelNo(someT: [String: Int], st:String) -> (name:String,wheels:Int) {
    return (st, someT[st as String]!)
}

let checkWheels = returnWheelNo(numberOfWheels, "bike")

println ( "the vehicle type \(checkWheels.name) has \(checkWheels.wheels) wheels")
```

Swift control flow statements: *if*

if statements look like C, but **curly brackets** are compulsory:

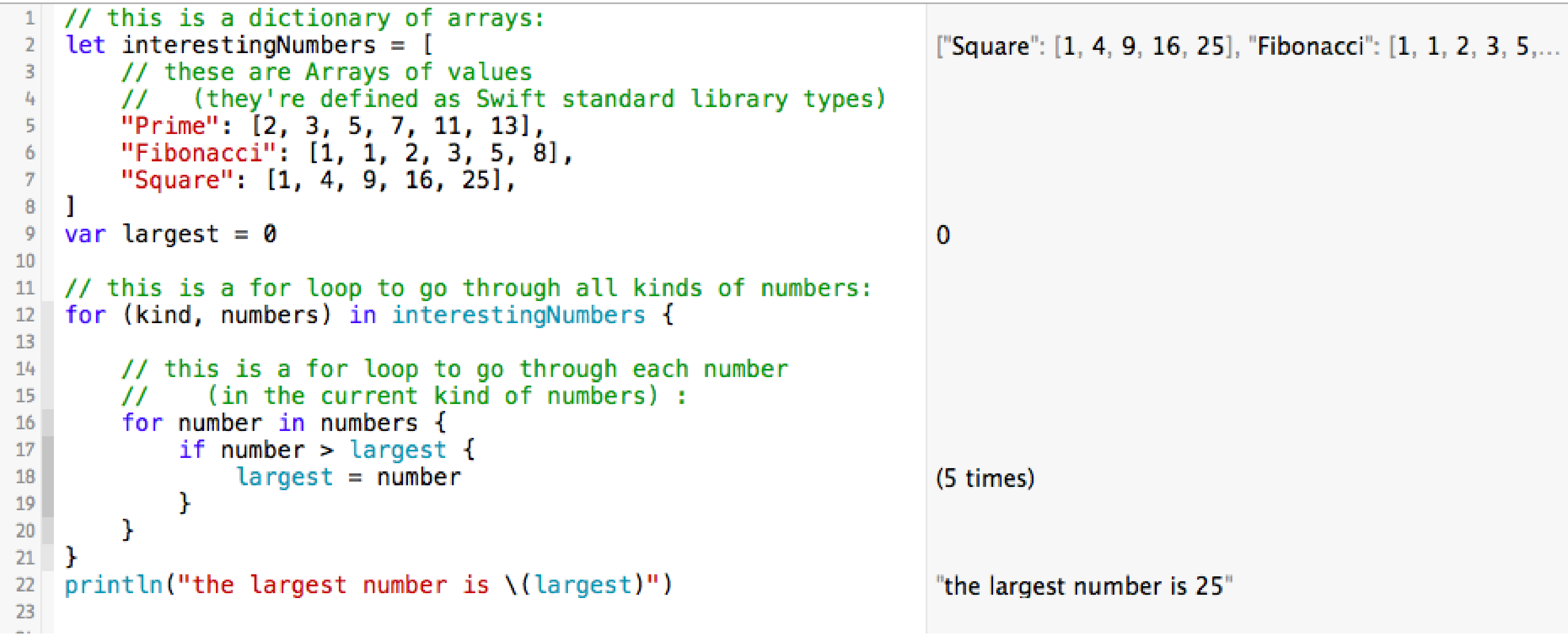


```
import UIKit

var str = "Hello, playground"

let studentsInWaitlist = 2

if (studentsInWaitlist > 1) {
    println("please extend enrollment limit")
} else {
    println("everybody is in")
}
```

```
"Hello, playground"

2

"please extend enrollmen...
```

Swift control flow statements: *for* loops

for loops in Swift can be written similarly to Python:

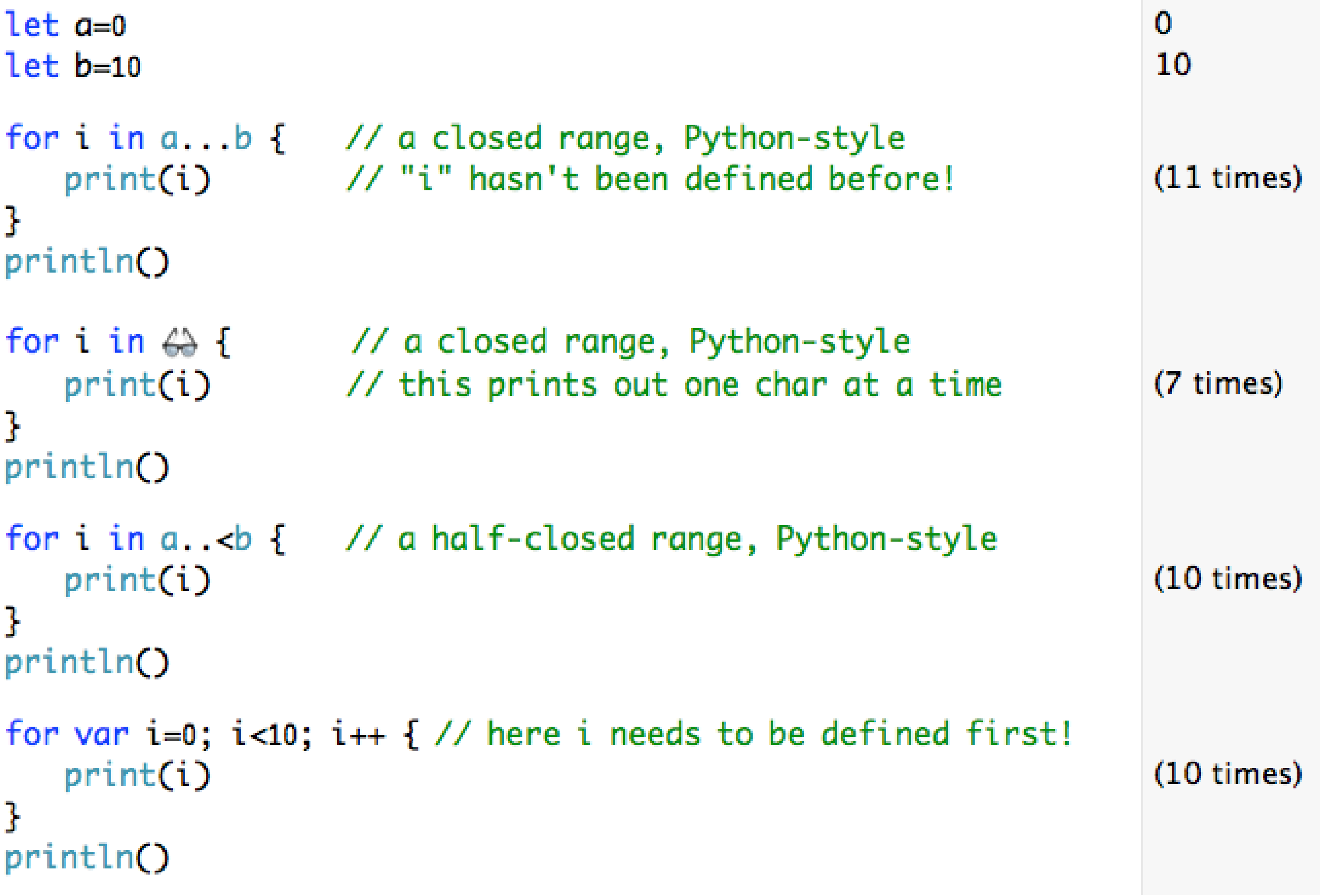


```
// this is a dictionary of arrays:
let interestingNumbers = [
    // these are Arrays of values
    //   (they're defined as Swift standard library types)
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0

// this is a for loop to go through all kinds of numbers:
for (kind, numbers) in interestingNumbers {

    // this is a for loop to go through each number
    //    (in the current kind of numbers) :
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
println("the largest number is \(largest)")
```

```
["Square": [1, 4, 9, 16, 25], "Fibonacci": [1, 1, 2, 3, 5,...

0

(5 times)

"the largest number is 25"
```

Swift *for loops*

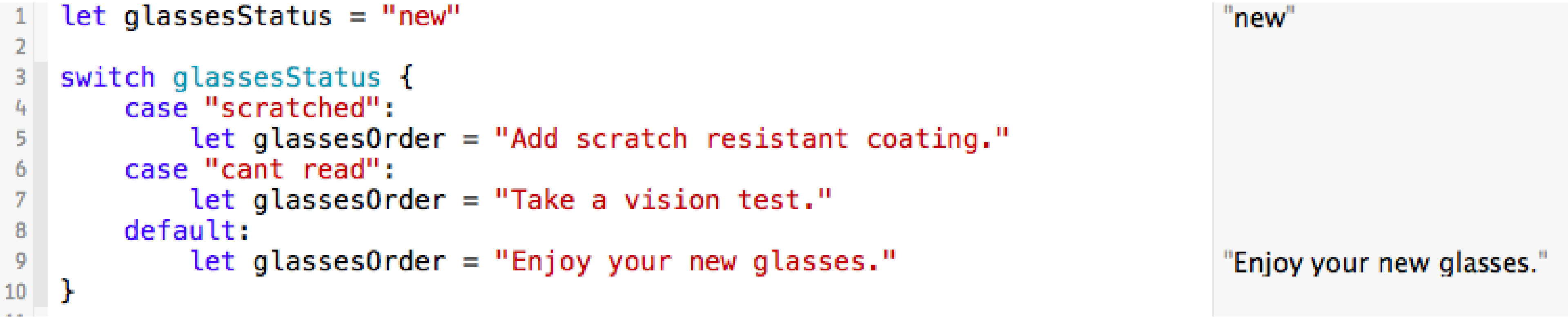
for loops can be Python-like with ranges, or C-like with 3 declarations:



```
let a=0                                                          0
let b=10                                                         10

for i in a...b {    // a closed range, Python-style
    print(i)        // "i" hasn't been defined before!           (11 times)
}
println()

for i in 👓 {        // a closed range, Python-style
    print(i)        // this prints out one char at a time        (7 times)
}
println()

for i in a..<b {    // a half-closed range, Python-style
    print(i)                                                     (10 times)
}
println()

for var i=0; i<10; i++ { // here i needs to be defined first!
    print(i)                                                     (10 times)
}
println()
```

Swift control flow statements: *switch/case*

switch/case statements, C-like, but a bit different in Swift:



```
let glassesStatus = "new"

switch glassesStatus {
    case "scratched":
        let glassesOrder = "Add scratch resistant coating."
    case "cant read":
        let glassesOrder = "Take a vision test."
    default:
        let glassesOrder = "Enjoy your new glasses."
}
```

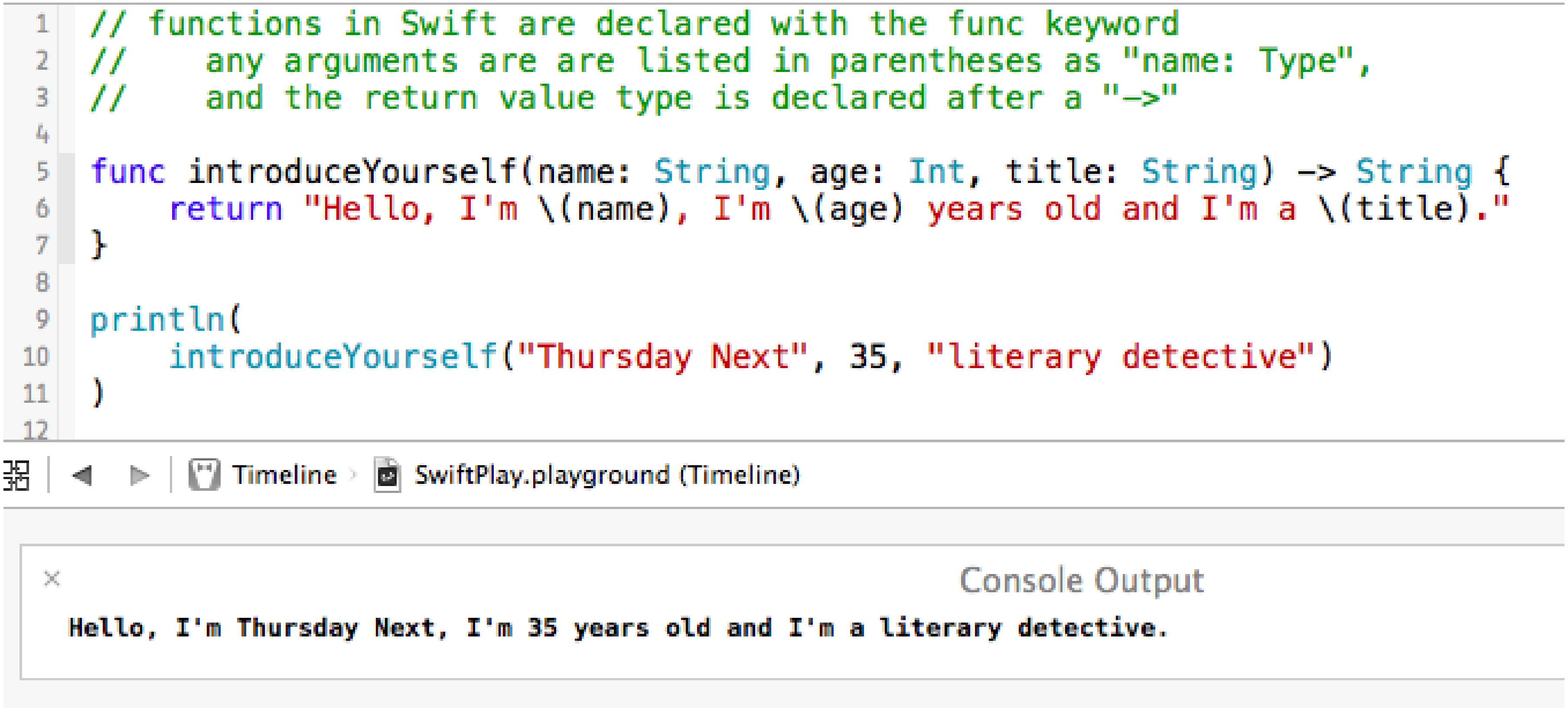
```
"new"
"Enjoy your new glasses."
```

the switch statement *must* be exhaustive!

the switch statement does *not* "Spring through"

Swift functions

functions in Swift have *input* arguments and output *return* values declared thus:



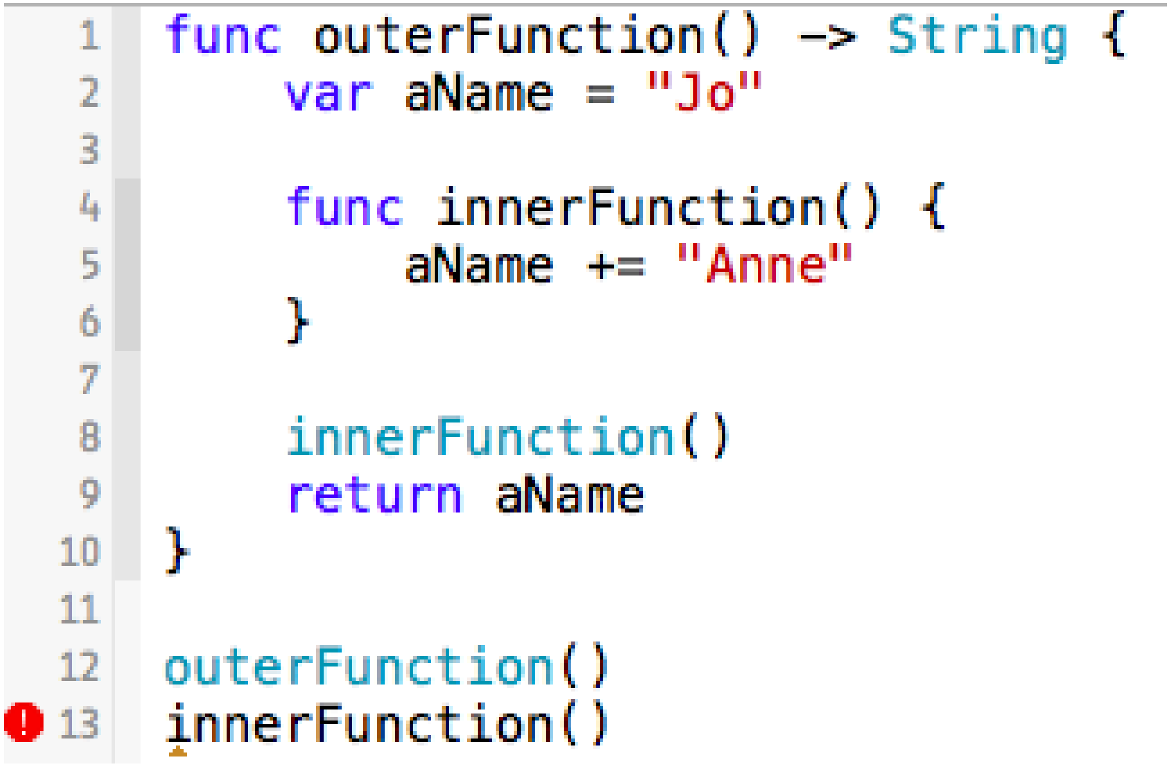
```swift
// functions in Swift are declared with the func keyword
//    any arguments are are listed in parentheses as "name: Type",
//    and the return value type is declared after a "->"

func introduceYourself(name: String, age: Int, title: String) -> String {
    return "Hello, I'm \(name), I'm \(age) years old and I'm a \(title)."
}

println(
    introduceYourself("Thursday Next", 35, "literary detective")
)

```

```
Hello, I'm Thursday Next, I'm 35 years old and I'm a literary detective.
```

Swift functions

functions in Swift can be *nested*! That means that you can define a function inside a function, and the inner function will be available only to code inside the outer function...

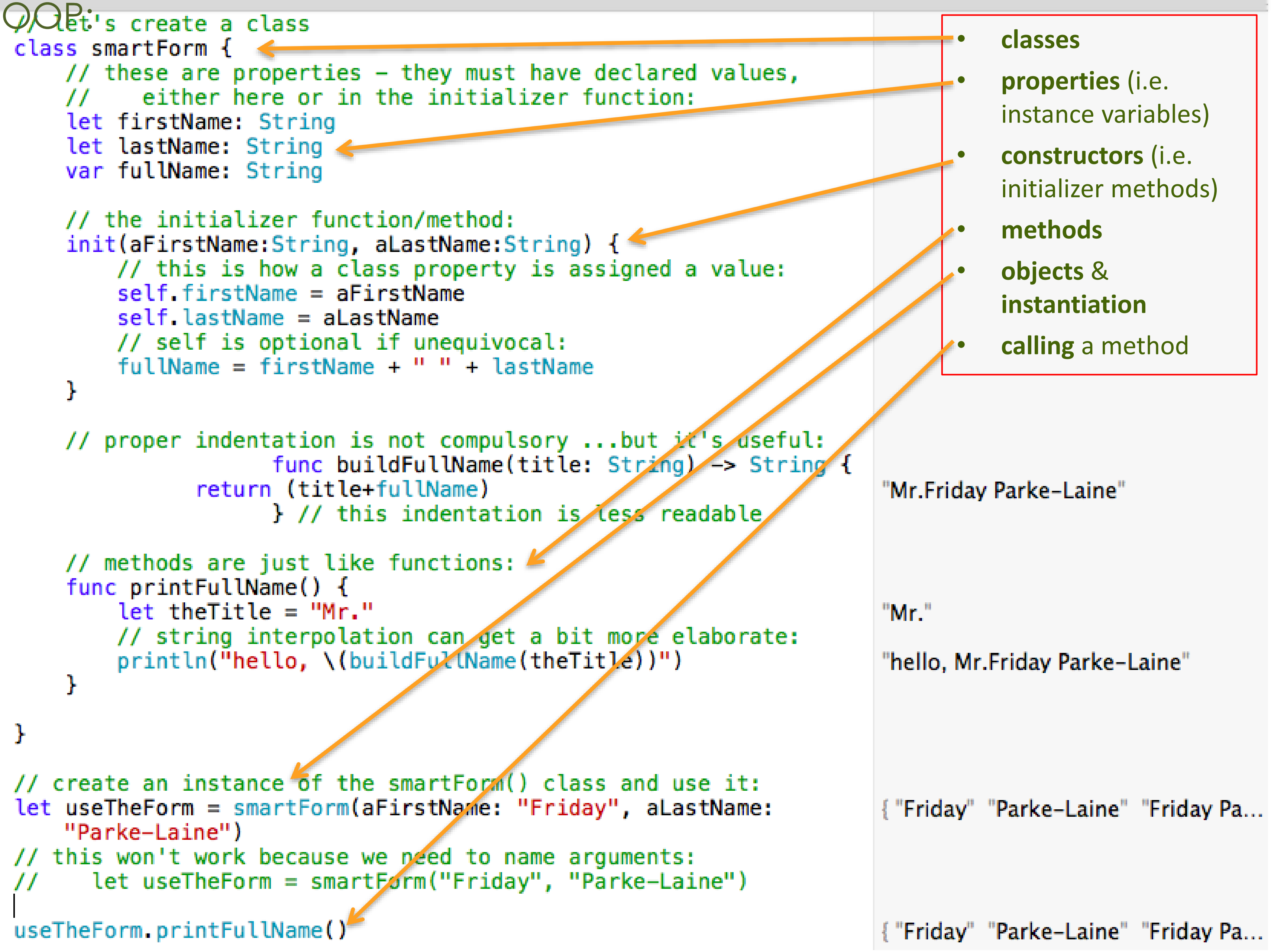
...maybe an example will make it clearer :-)



```swift
func outerFunction() -> String {
    var aName = "Jo"

    func innerFunction() {
        aName += "Anne"
    }

    innerFunction()
    return aName
}

outerFunction()
innerFunction()
```

Swift is an Object Oriented language...

review the following terms and make sure you remember the basics of OOP:



```
// let's create a class
class smartForm {
    // these are properties - they must have declared values,
    //    either here or in the initializer function:
    let firstName: String
    let lastName: String
    var fullName: String

    // the initializer function/method:
    init(aFirstName:String, aLastName:String) {
        // this is how a class property is assigned a value:
        self.firstName = aFirstName
        self.lastName = aLastName
        // self is optional if unequivocal:
        fullName = firstName + " " + lastName
    }

    // proper indentation is not compulsory ...but it's useful:
                func buildFullName(title: String) -> String {
          return (title+fullName)
                } // this indentation is less readable

    // methods are just like functions:
    func printFullName() {
        let theTitle = "Mr."
        // string interpolation can get a bit more elaborate:
        println("hello, \(buildFullName(theTitle))")
    }

}

// create an instance of the smartForm() class and use it:
let useTheForm = smartForm(aFirstName: "Friday", aLastName:
    "Parke-Laine")
// this won't work because we need to name arguments:
//    let useTheForm = smartForm("Friday", "Parke-Laine")
|
useTheForm.printFullName()
```

```
"Mr.Friday Parke-Laine"

"Mr."

"hello, Mr.Friday Parke-Laine"

{ "Friday" "Parke-Laine" "Friday Pa...

{ "Friday" "Parke-Laine" "Friday Pa...
```

- **classes**
- **properties** (i.e. instance variables)
- **constructors** (i.e. initializer methods)
- **methods**
- **objects** & **instantiation**
- **calling** a method

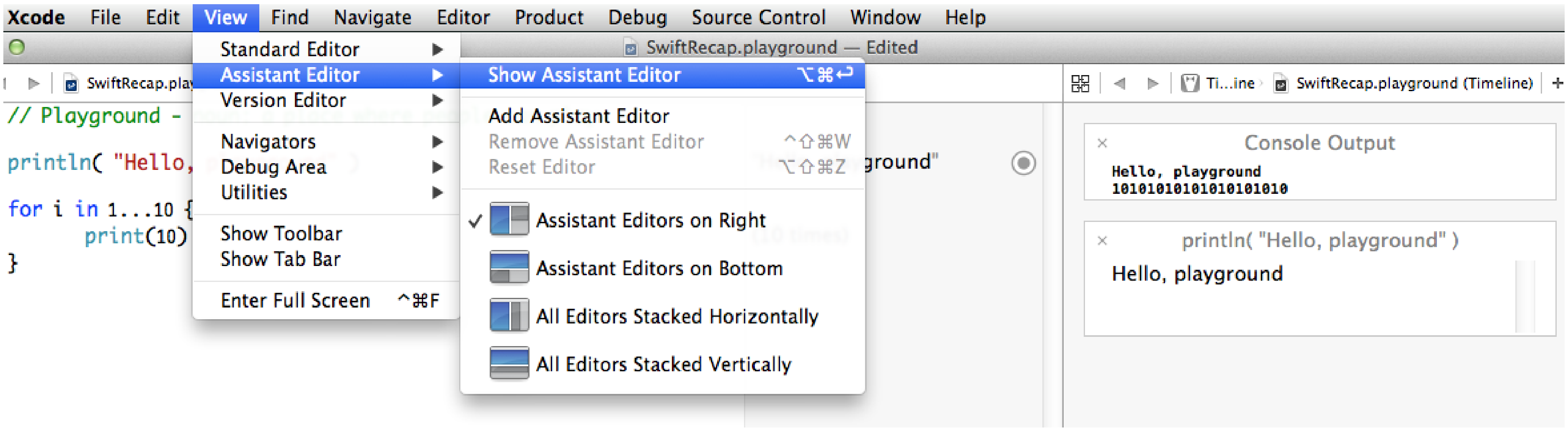
Note: where is the *output* for Swift Playgrounds?

when testing Swift code in Xcode, Playgrounds are useful...

...it's useful to view the output of *print/println* statements too (which is *not the same* as code evaluation results).

You can activate Playground's "Console Output" thus:

select the menu *View →Assistant Editor →Show Assistant Editor*



```
// Playground -
println( "Hello,
for i in 1...10 {
        print(10)
}
```

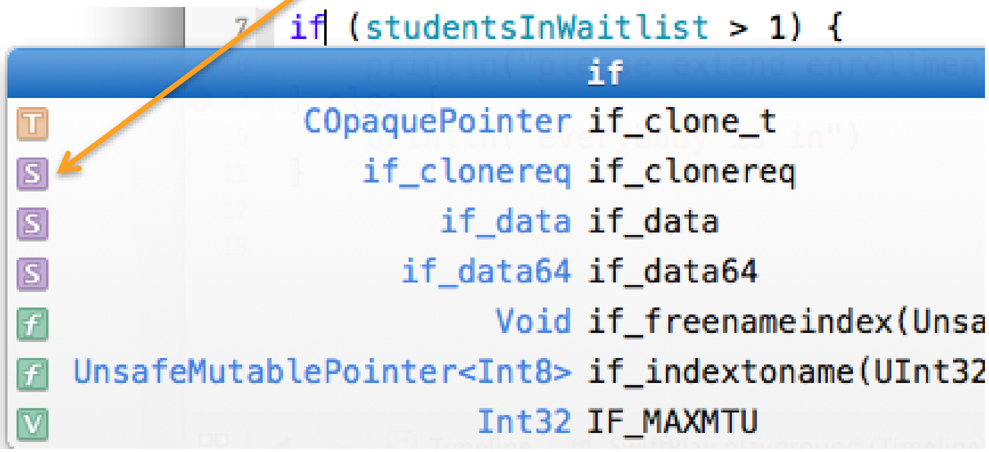
```
Hello, playground
10101010101010101010
```

```
Hello, playground
```

Note: Xcode "Code Sense" autocompletion

what are these colored squares and what do the letters mean?

"Code Sense" is the name of Xcode's autocompletion facility.



```
if| (studentsInWaitlist > 1) {
                              if
T      COpaquePointer if_clone_t
S         if_clonereq if_clonereq
S             if_data if_data
S           if_data64 if_data64
f                Void if_freenameindex(Unsa
f UnsafeMutablePointer<Int8> if_indextoname(UInt32
V               Int32 IF_MAXMTU
```

Red: macros

- # = macro (think `#define`)

Brown: Core Data / namespace

- C = modeled **class**
- M = modeled **method**
- P = modeled **property**

- N = C++ **namespace**

Orange: aliased types

- C̲ = Objective-C **category**
- E = **enum**
- T = **typedef**

Green: variables

- B = **binding**
- *ƒ* = **function**
- F = **field**
- K = **constant**
- L = **local variable**
- O = IB**Outlet**
- V = **variable** (can be ivar, global var, local var, etc.)
- *x* = parameter (think *f(x)*)

Blue: methods

- A = IB**Action**
- M = **method**
- P = **property**

Purple: aggregate types

- C = **class** (Objective-C or C++)
- ₵ = **class extension**
- Pr = Objective-C **protocol**
- S = **struct**
- U = **union**

Lab 07:

1. in the *switch/case* example, try removing the default case, and change the value of *glassesStatus* until you get an error.
   Write down the error type you get in a *comment* below your code.

2. write a Swift function named *twoThings* that takes two floating-point arguments, and returns both their *sum* and their *product* as a 2-value ***tuple***. Then provide two examples of calling the
   twoThings() function you just wrote. (see above slides for a Swift function example, and about tuples)

3. in the *for* loop example, define and use another variable to keep track of which *kind* of number – "Prime", "Fibonacci", or "Square" – was the largest, and print down the result. (the largest *value* is already computed in the inner *for* loop in that example)

Save all the above in one single playground named "lab07username.playground" and turn it in to your IU GitHub repository.