



SWIFT – THE ESSENTIALS

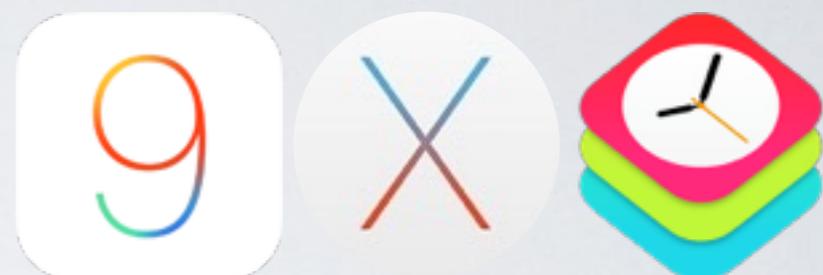
Nick Chen
Fall 2015
talentspark.io

SYLLABUS

Week 1	Swift 2.0, Practical Xcode
Week 2	First iOS app and TableViews
Week 3	TableViews and Auto Layout
Week 4	More TableViews with Parse SDK
Week 5	Animation
Week 6	Core Location & Mapkit
Week 7	<i>Project</i>
Week 8	<i>Project</i>

INTRO

- New programming language for iOS, OS X and watch OS.
- Swift 1 introduced in WWDC 2014.
Swift 2 introduced in WWDC 2015.
- “...drawing ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list.”



REFERENCE

The Swift Programming Language (Swift 2 Prerelease)

Table of Contents

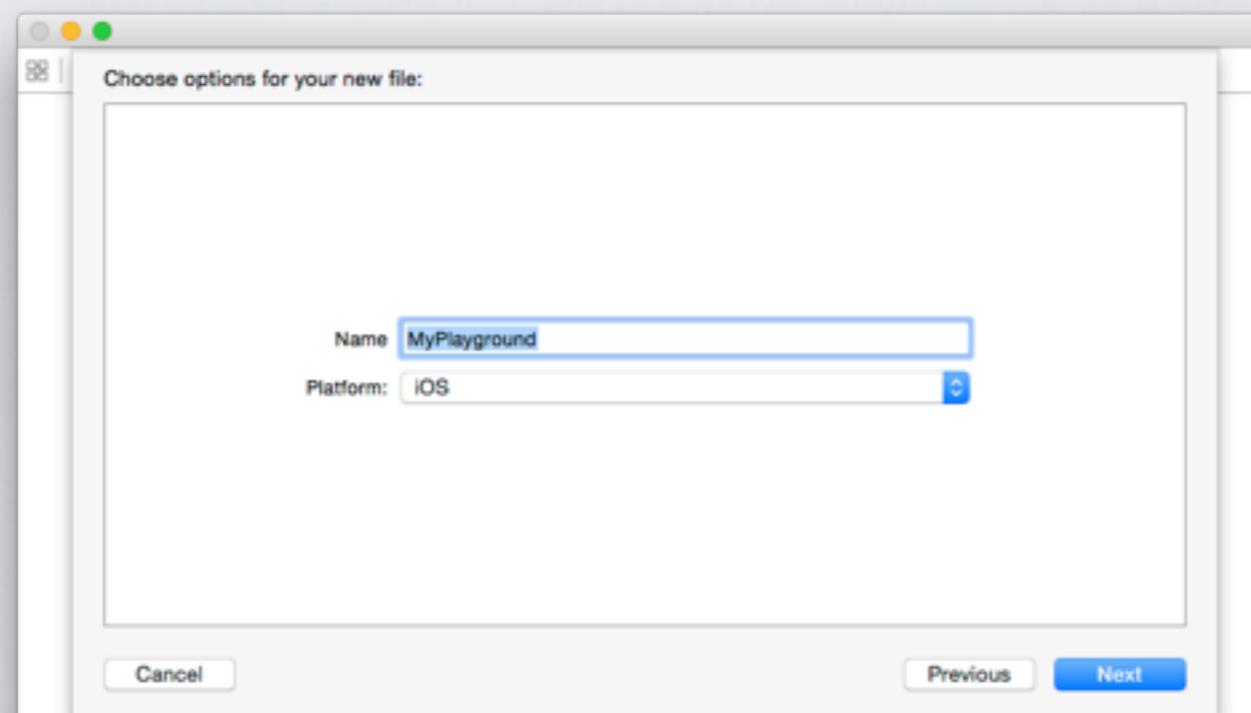
Welcome to Swift	1
About Swift	2
A Swift Tour	5
Language Guide	48
The Basics	49
Basic Operators	98
Strings and Characters	126
Collection Types	163
Control Flow	205
Functions	263
Closures	302
Enumerations	326

Click to page 870

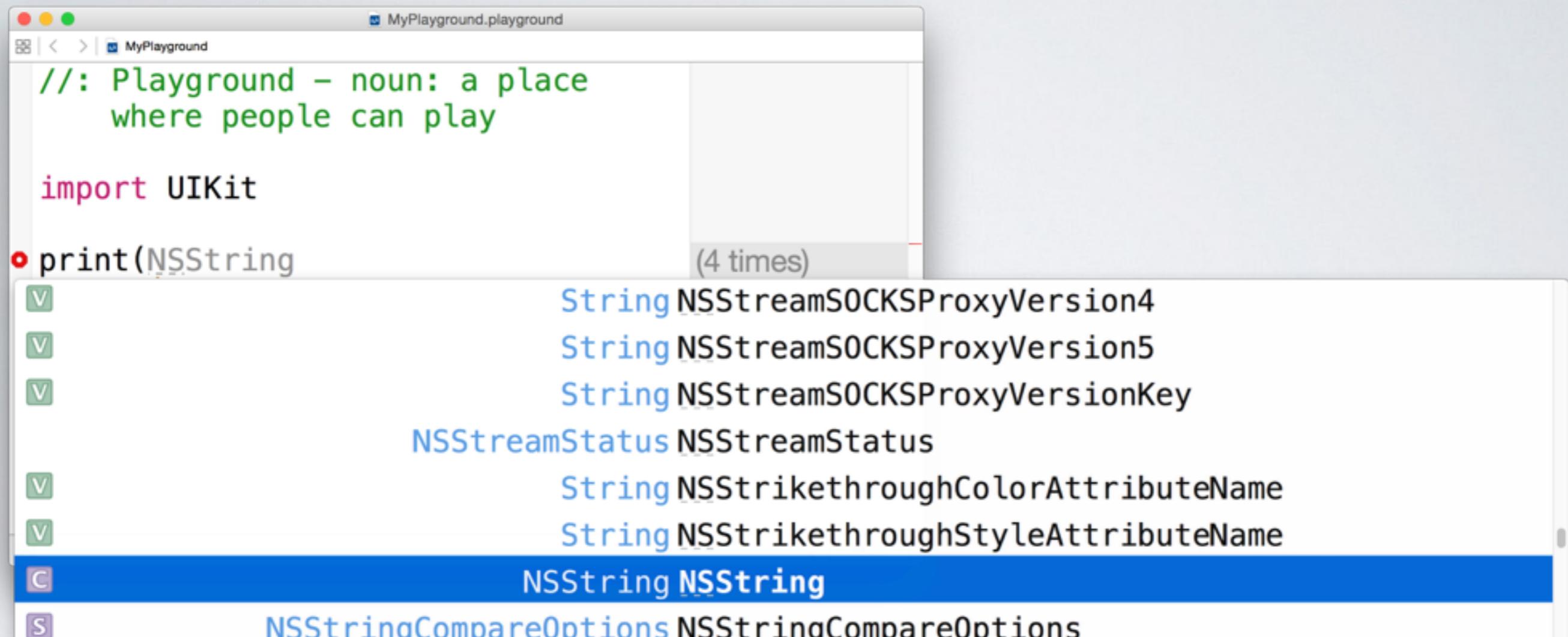
Page 1



PLAYGROUNDS



PLAYGROUNDS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code being typed is:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
print(NSString
```

The code completion dropdown is open, showing suggestions for the `NSString` class. The suggestions are:

- (4 times)
 - String `NSStreamSOCKSProxyVersion4`
 - String `NSStreamSOCKSProxyVersion5`
 - String `NSStreamSOCKSProxyVersionKey`
- `NSStreamStatus` `NSStreamStatus`
- V `NSStrikethroughColorAttributeName`
- V `NSStrikethroughStyleAttributeName`
- C `NSString` **`NSString`**
- S `NSStringCompareOptions` `NSStringCompareOptions`

The `NSString` class declares the programmatic interface for an object that manages immutable strings. An immutable string is a text string that is defined when it is created and subsequently cannot be changed. `NSString` is implemented to represent an array of Unicode characters, in other words, a text string. [More...](#)

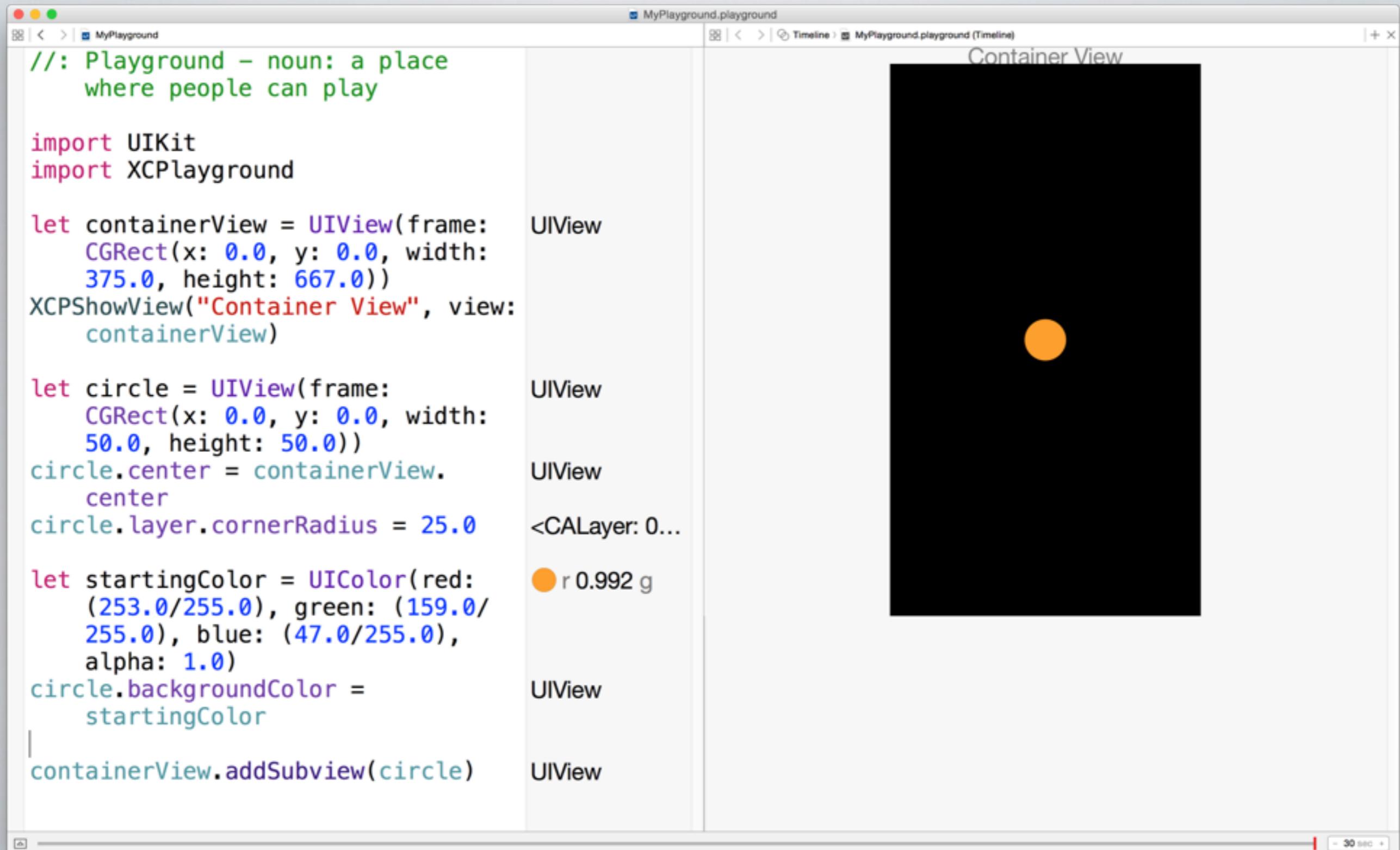
PLAYGROUNDS

The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code area contains the following:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
! print(a)    ! Use of unresolved identifier 'a'
```

A red error bar highlights the line `! print(a)`, indicating an unresolved identifier 'a'. The status bar at the bottom right shows a timer set to "30 sec".

PLAYGROUNDS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The playground contains the following Swift code:

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

import UIKit
import XCPlayground

let containerView = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 375.0, height: 667.0))
XCPShowView("Container View", view: containerView)

let circle = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 50.0, height: 50.0))
circle.center = containerView.center
circle.layer.cornerRadius = 25.0

let startingColor = UIColor(red: (253.0/255.0), green: (159.0/255.0), blue: (47.0/255.0), alpha: 1.0)
circle.backgroundColor = startingColor

containerView.addSubview(circle)
```

The playground output pane shows the types of each variable and the final state of the circle's color. The preview pane shows a black container view with a single orange circle centered in it.

HELLO, WORLD!

A screenshot of an Xcode playground window titled "MyPlayground.playground". The playground contains the following Swift code:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
print("Hello, World!")
```

The output pane on the right shows the result of the print statement: "Hello, World!".

The bottom of the window shows a toolbar with a play button, a progress bar, and a timer set to "30 sec".

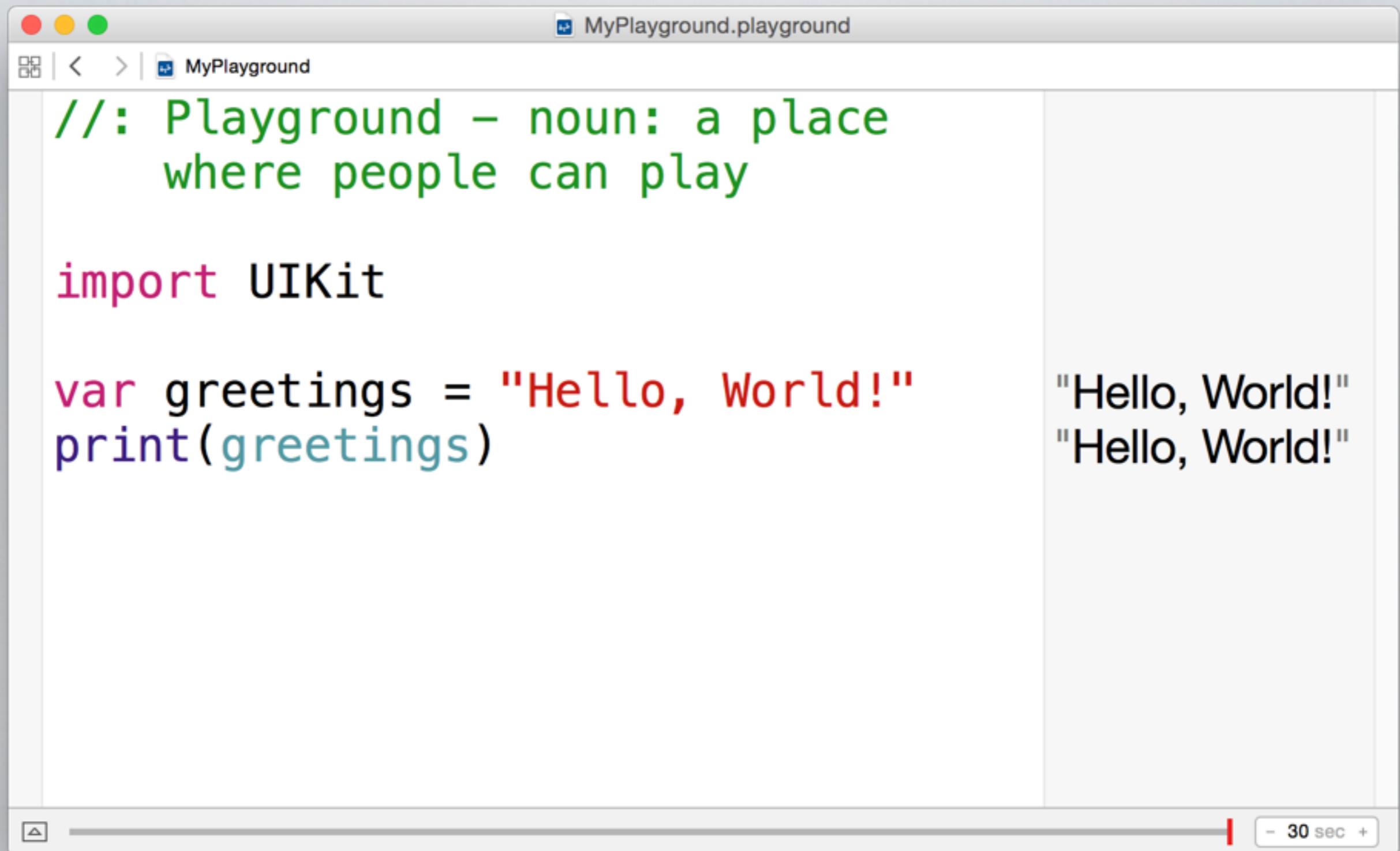
HELLO, WORLD!

A screenshot of an Xcode playground window titled "MyPlayground.playground". The playground contains the following Swift code:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
print("Hello, World!")
```

The output pane shows the result of the `print` statement: "Hello, World!". There is also a preview pane showing a small card with the text "Hello, World!". The bottom right corner of the preview pane has a timer set to "30 sec".

VARIABLES VS CONSTANTS

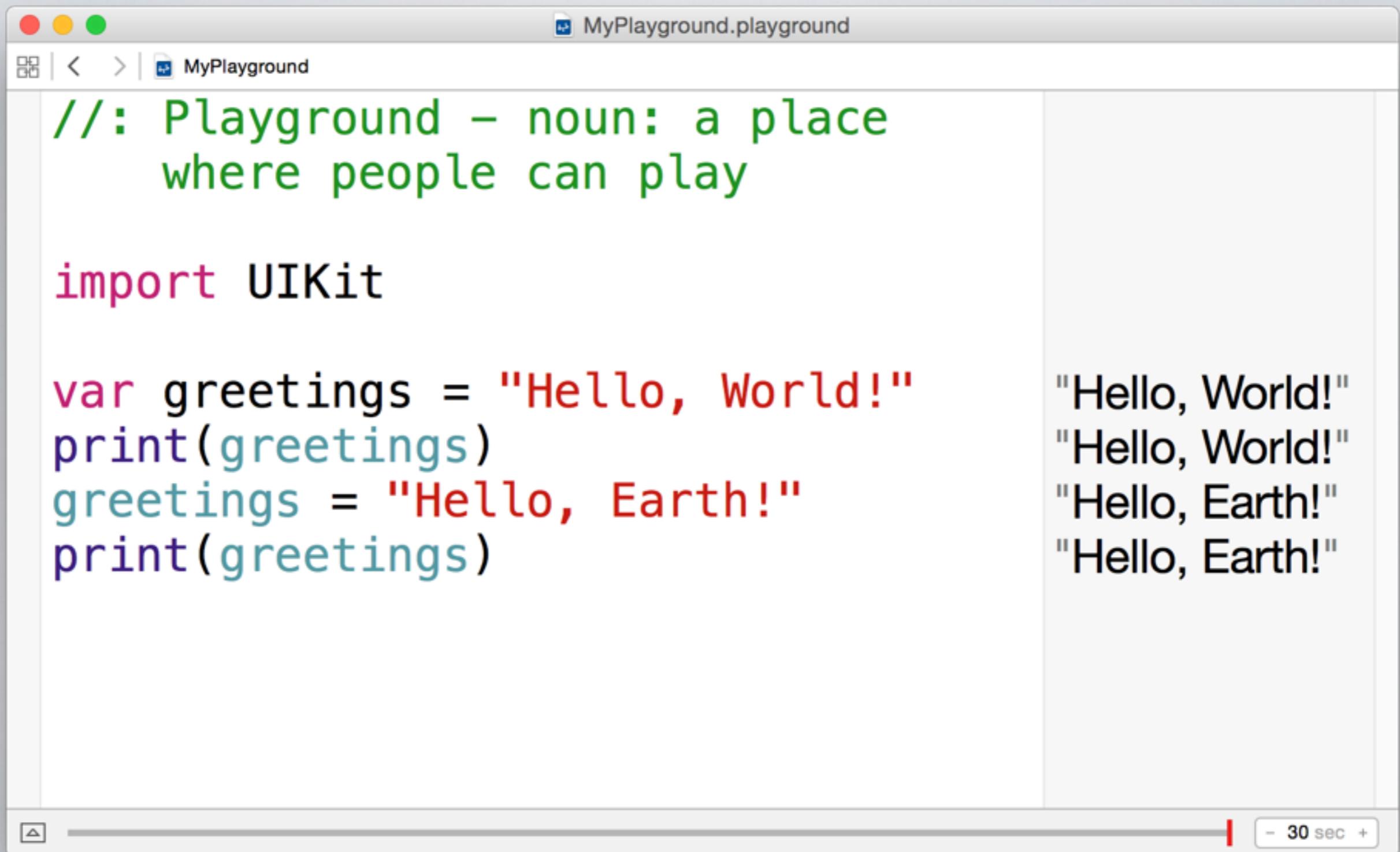


A screenshot of an Xcode playground window titled "MyPlayground.playground". The playground contains the following Swift code:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
var greetings = "Hello, World!"  
print(greetings)
```

The output pane shows two lines of text: "Hello, World!" and "Hello, World!". At the bottom of the window, there is a toolbar with a search field containing "- 30 sec +".

VARIABLES VS CONSTANTS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

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

import UIKit

var greetings = "Hello, World!"
print(greetings)
greetings = "Hello, Earth!"
print(greetings)
```

The output pane displays the results of the playground's execution:

Output	Description
"Hello, World!"	Initial value assigned to the variable.
"Hello, World!"	Value printed after the first print statement.
"Hello, Earth!"	Value assigned to the variable via an assignment statement.
"Hello, Earth!"	Value printed after the second print statement.

The bottom right corner of the playground window shows a timer set to "- 30 sec +".

VARIABLES VS CONSTANTS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code is as follows:

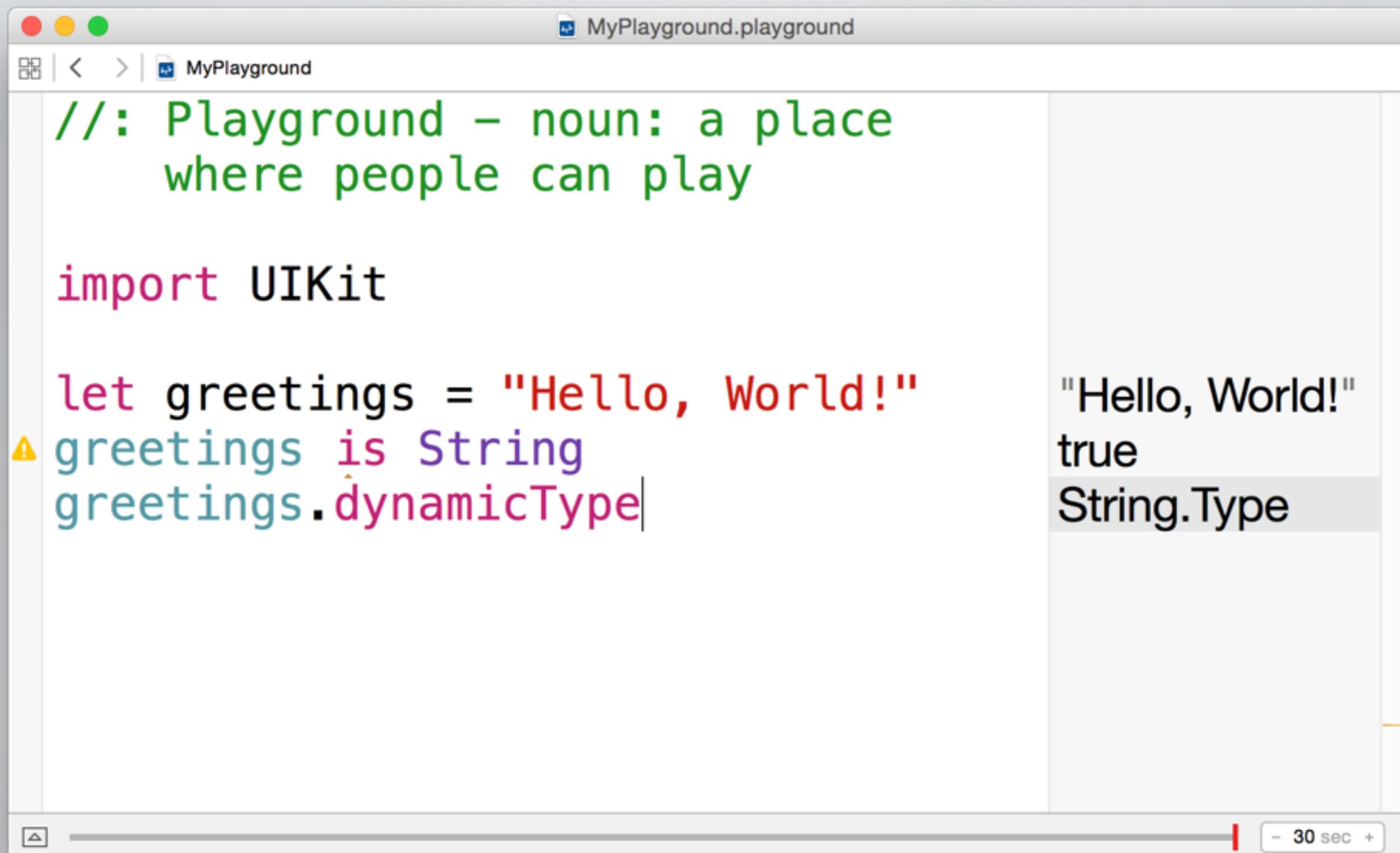
```
//: Playground - noun: a place where people can play

import UIKit

let greetings = "Hello, World!"
print(greetings)
• greetings = Cannot assign to value: 'greetings' is a 'let' constant
Fix-it Change 'let' to 'var' to make it mutable
print(greetings)
```

The code demonstrates the use of a constant (let) and a mutable variable (var). The first assignment is successful, printing "Hello, World!". The second assignment, where the constant is re-assigned, fails with the error message "Cannot assign to value: 'greetings' is a 'let' constant". A "Fix-it" suggestion is provided: "Change 'let' to 'var' to make it mutable". When the code is run again, it prints "Hello, Earth!".

PRIMITIVE TYPES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code editor contains the following Swift code:

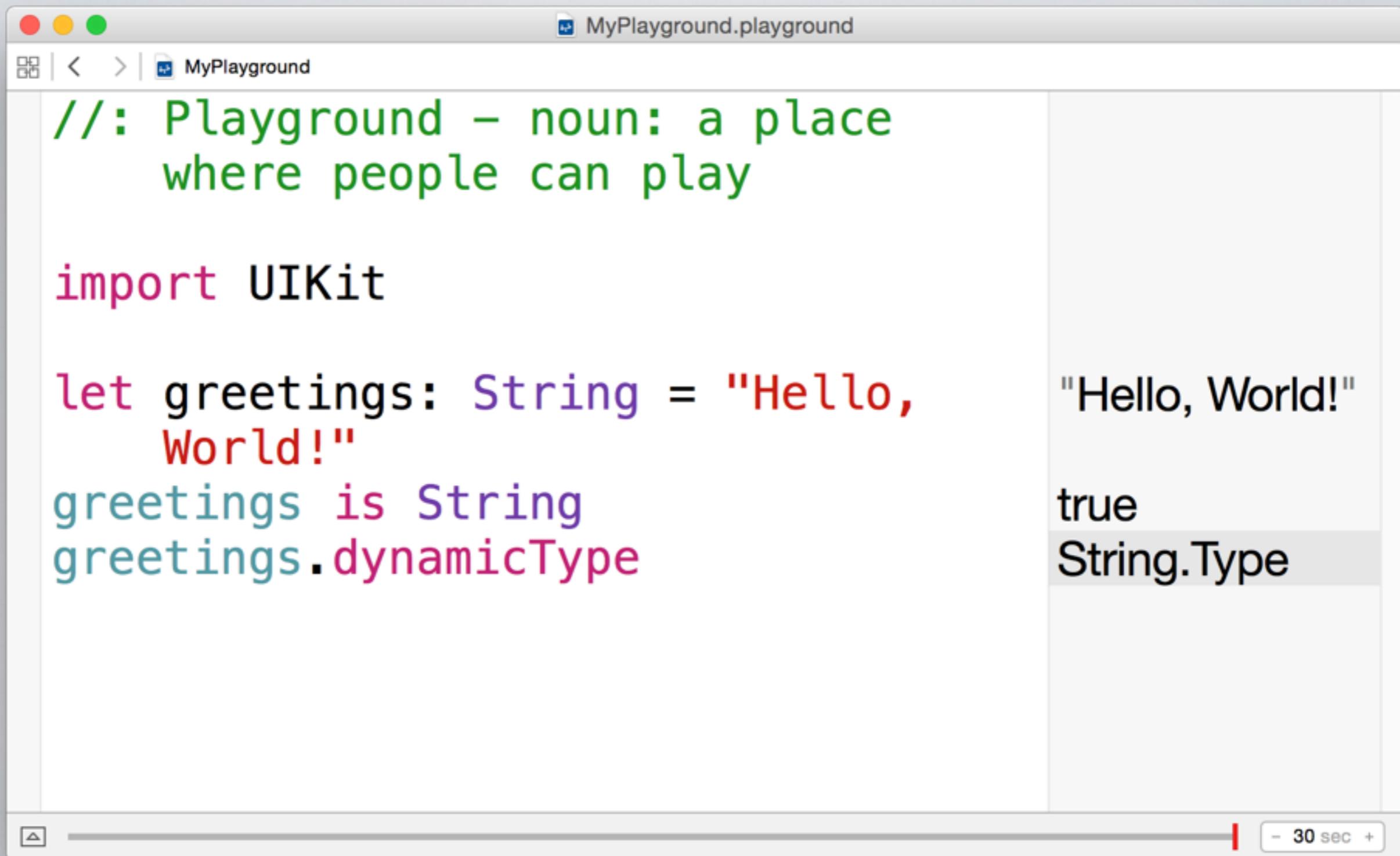
```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
let greetings = "Hello, World!"  
⚠️ greetings is String  
greetings.dynamicType
```

The playground's output pane shows the results of the code execution:

- "Hello, World!"
- true
- String.Type

The "dynamicType" method is highlighted with a yellow warning icon.

PRIMITIVE TYPES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

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

import UIKit

let greetings: String = "Hello, World!"
greetings is String
greetings.dynamicType
```

The playground shows the output of the code on the right side of the interface. The first two lines of code ("Hello, World!" and "true") are displayed in black text. The third line of code, "greetings.dynamicType", has a dropdown menu open, showing the type "String.Type" highlighted in grey.

PRIMITIVE TYPES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The playground contains two code snippets demonstrating primitive types.

Code Snippet 1:

```
//: Playground - noun: a place  
// where people can play  
  
import UIKit  
  
let myNumber = 1234  
myNumber.dynamicType
```

Output 1:

```
1234  
Int.Type
```

Code Snippet 2:

```
// You probably won't need to use  
// Int16, Int32, Int64  
// UInt16, UInt32, UInt64  
let myUInt16: UInt16 = 1234  
myNumber.dynamicType
```

Output 2:

```
1234  
Int.Type
```

PRIMITIVE TYPES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code demonstrates the dynamic nature of floating-point types:

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

import UIKit

let myFloatingPoint = 1.234
myFloatingPoint.dynamicType
```

1.234
Double.Type

```
let anotherFloatingPoint: Float = 1.234
anotherFloatingPoint.dynamicType
```

1.234
Float.Type

PRIMITIVE TYPES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code demonstrating primitive types is as follows:

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

import UIKit

let myBoolean = true
myBoolean.dynamicType
```

The output for the first two lines is:

```
true  
Bool.Type
```

The code then attempts to use an integer as a protocol conformer:

```
// Does not let you do this
// unlike other languages
let i = 1
if i {
```

A red error message box highlights the opening brace of the if statement with the text:

! Type 'Int' does not conform to protocol 'B...'

At the bottom right of the playground window, there is a timer set to "30 sec".

STRING INTERPOLATION

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

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

import UIKit

let myInt = 2
let myBool = false
print("The value of myInt is \(myInt) and the value of myBool is \(myBool)")
```

The output of the print statement is displayed in a callout box at the bottom left:

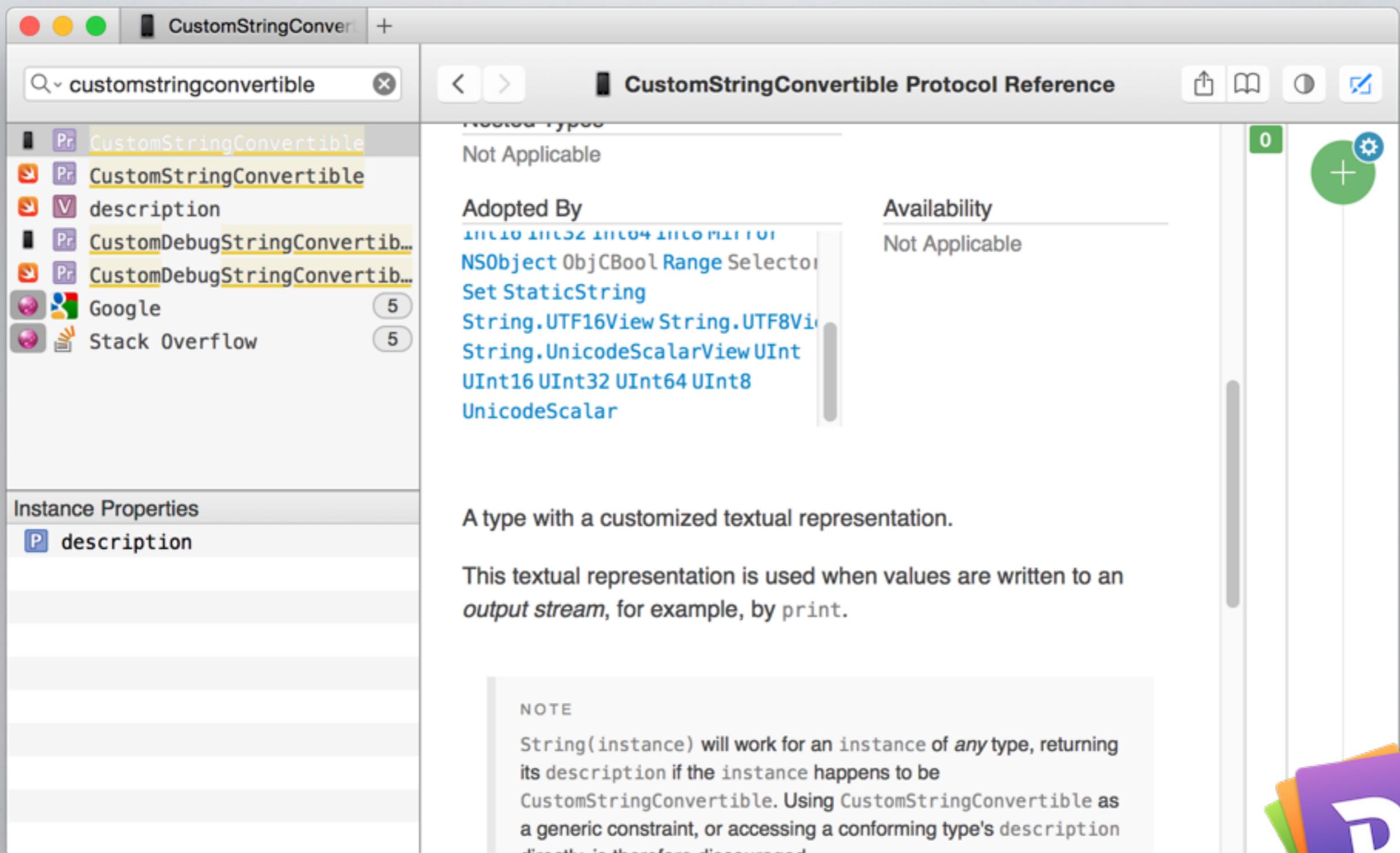
The value of myInt is 2 and the value of myBool is false

On the right side of the playground, the individual values are listed:

- 2
- false
- "The valu..." (with a circular disclosure button)

At the bottom right of the playground window, there is a timer set to "30 sec".

STRING INTERPOLATION



The screenshot shows the Xcode documentation interface for the `CustomStringConvertible` protocol. The search bar at the top left contains the query `customstringconvertible`. The main content area is titled `CustomStringConvertible Protocol Reference`. The protocol is described as follows:

- Protocol Types:** Not Applicable
- Adopted By:** `NSObject`, `ObjCBool`, `Range`, `Selector`, `Set`, `StaticString`, `String.UTF16View`, `String.UTF8View`, `String.UnicodeScalarView`, `UInt`, `UInt16`, `UInt32`, `UInt64`, `UInt8`, `UnicodeScalar`
- Availability:** Not Applicable

Instance Properties:

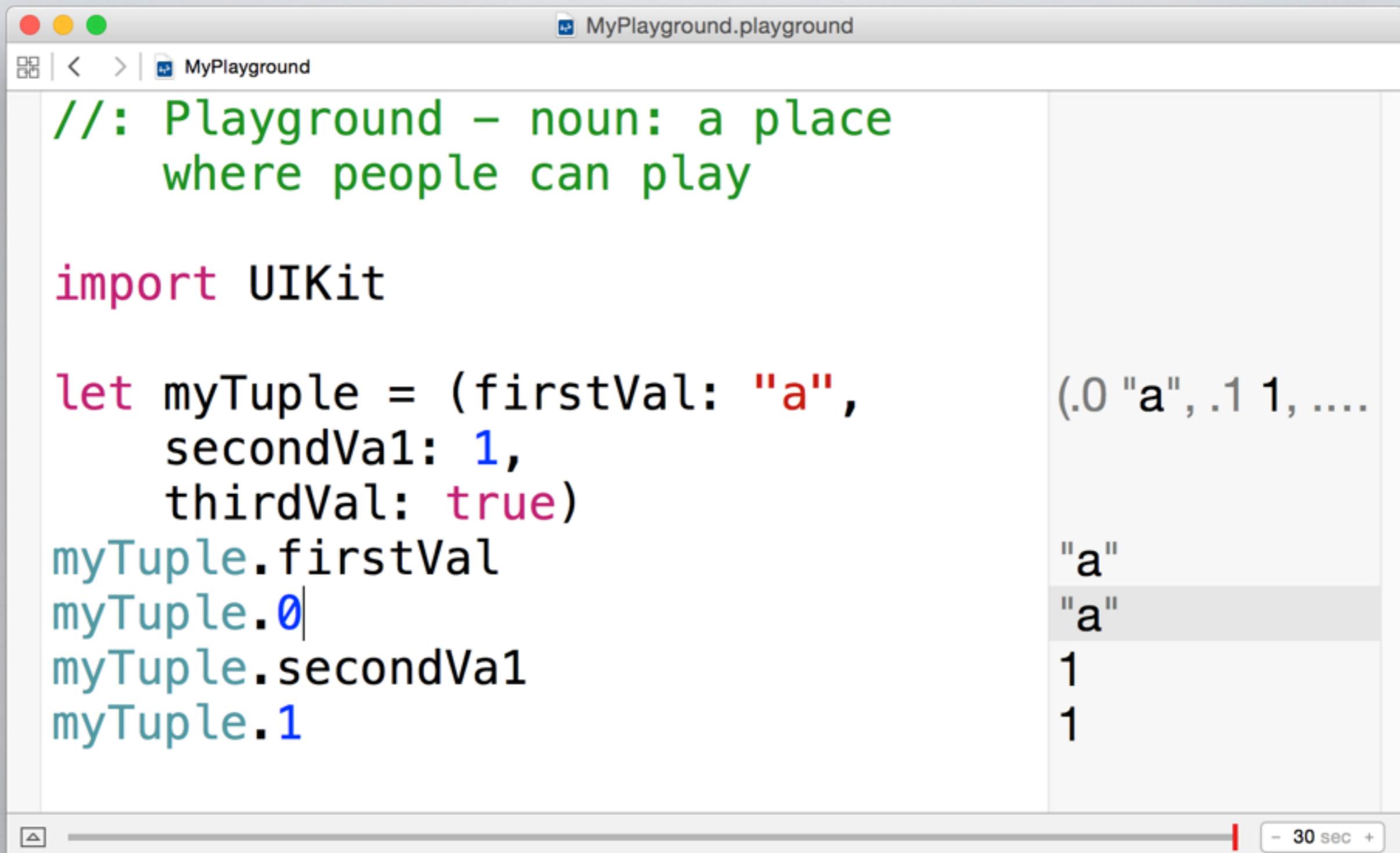
- `P description`

A note states: "A type with a customized textual representation. This textual representation is used when values are written to an *output stream*, for example, by `print`".

NOTE: `String(instance)` will work for an instance of *any* type, returning its description if the instance happens to be `CustomStringConvertible`. Using `CustomStringConvertible` as a generic constraint, or accessing a conforming type's description directly, is therefore discouraged.



COLLECTIONS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

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

import UIKit

let myTuple = (firstVal: "a",
               secondVal: 1,
               thirdVal: true)
myTuple.firstVal
myTuple.0
myTuple.secondVal
myTuple.1
```

The playground output pane displays the results of the tuple operations:

(.0 "a", .1 1,
"a"
"a"
1
1

The output for "myTuple.0" is highlighted in blue, indicating it is being evaluated.

COLLECTIONS

A screenshot of an Xcode playground window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
import UIKit

// Array of type-inferred ints
var mutableArray = [1, 2, 3]
```

The variable declaration "mutableArray" is highlighted. A callout bubble shows the declaration "Declaration var mutableArray: [Int]" and the file "Declared In MyPlayground.playground". To the right of the code editor, the array value "[1, 2, 3]" is displayed in a preview pane.

Below the code editor, the array elements are listed:

mutableArray[0]	1
mutableArray[1]	2
mutableArray[2]	3
// You can change elements	
mutableArray[3] = 4	4

The bottom of the window shows a toolbar with a play button, a progress bar, and a timer set to "30 sec".

COLLECTIONS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code area contains the following:

```
// Array of type-inferred ints
let immutableArray = [1, 2, 3]
```

A tooltip below the declaration shows:

Declaration let immutableArray: [Int]
Declared In MyPlayground.playground

On the right side, the array is displayed as [1, 2, 3]. Below the array, three subscripts are shown:

```
immutableArray[0]
immutableArray[1]
immutableArray[2]
```

Below these, a comment and an attempt to assign to the fourth element are shown:

```
// You can't change elements
immutableArray[3] = 4
```

A red error message is displayed:

Cannot assign through subscript: 'immutableAr...' - 30 sec +

The status bar at the bottom right shows "- 30 sec +".

COLLECTIONS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code editor contains the following Swift code:

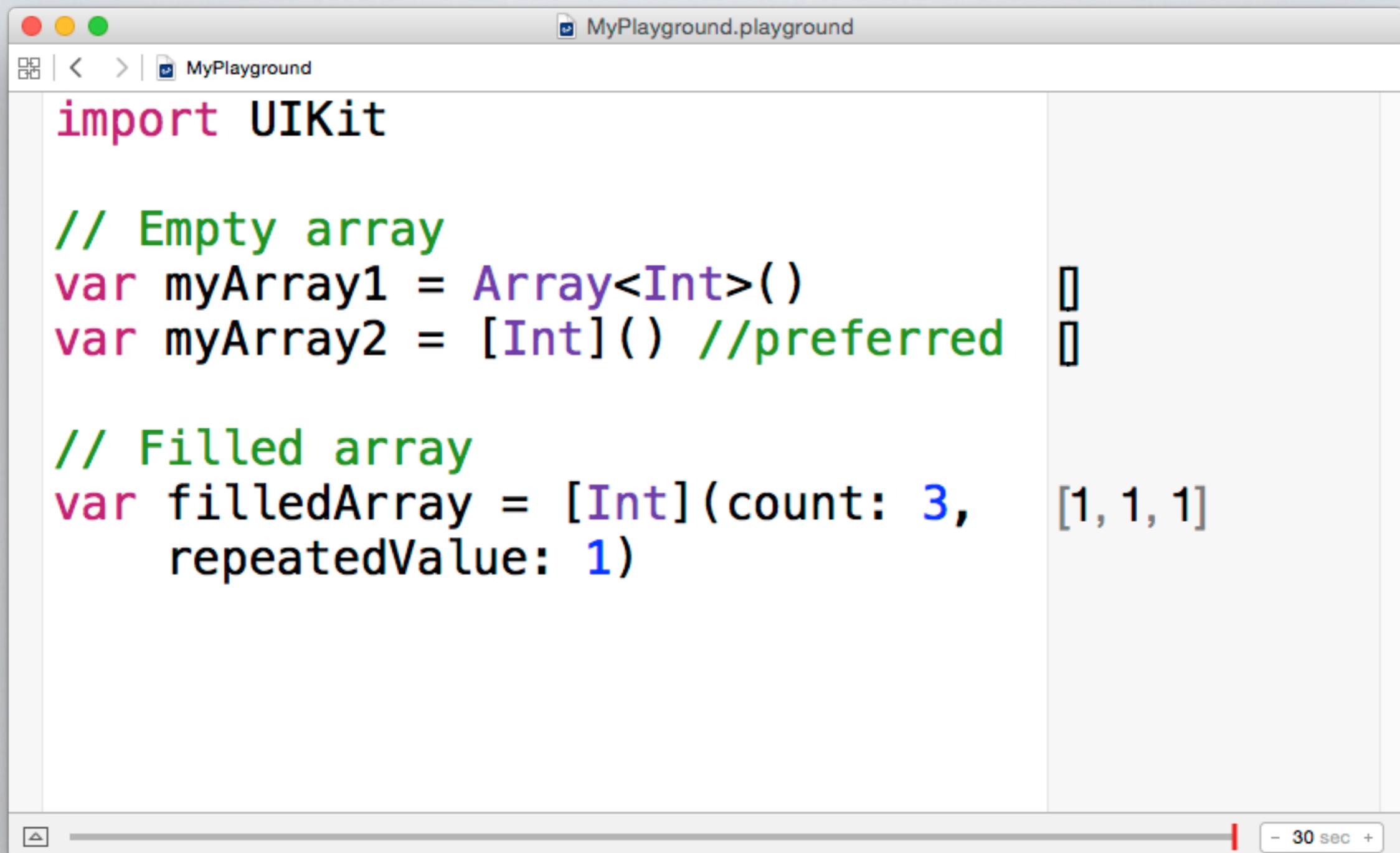
```
import UIKit

// This compiles
// but please don't mix types in
// array
let myArray = ["string", 1, true]
```

A tooltip for the declaration "let myArray" is displayed, showing its type as `[NSObject]` and that it is declared in `MyPlayground.playground`.

The playground output area shows the result of the execution: `["string", 1, 1]`. A timer at the bottom right indicates the execution time is 30 seconds.

COLLECTIONS



The screenshot shows a Mac OS X window titled "MyPlayground.playground". The main area contains Swift code demonstrating array creation:

```
import UIKit

// Empty array
var myArray1 = Array<Int>()
var myArray2 = [Int]() //preferred

// Filled array
var filledArray = [Int](count: 3, repeatedValue: 1)
```

The code uses two different ways to create arrays: `Array<Int>()` and `[]` (the empty array literal). It also shows how to create a filled array with three elements, all set to the value 1.

COLLECTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
import UIKit

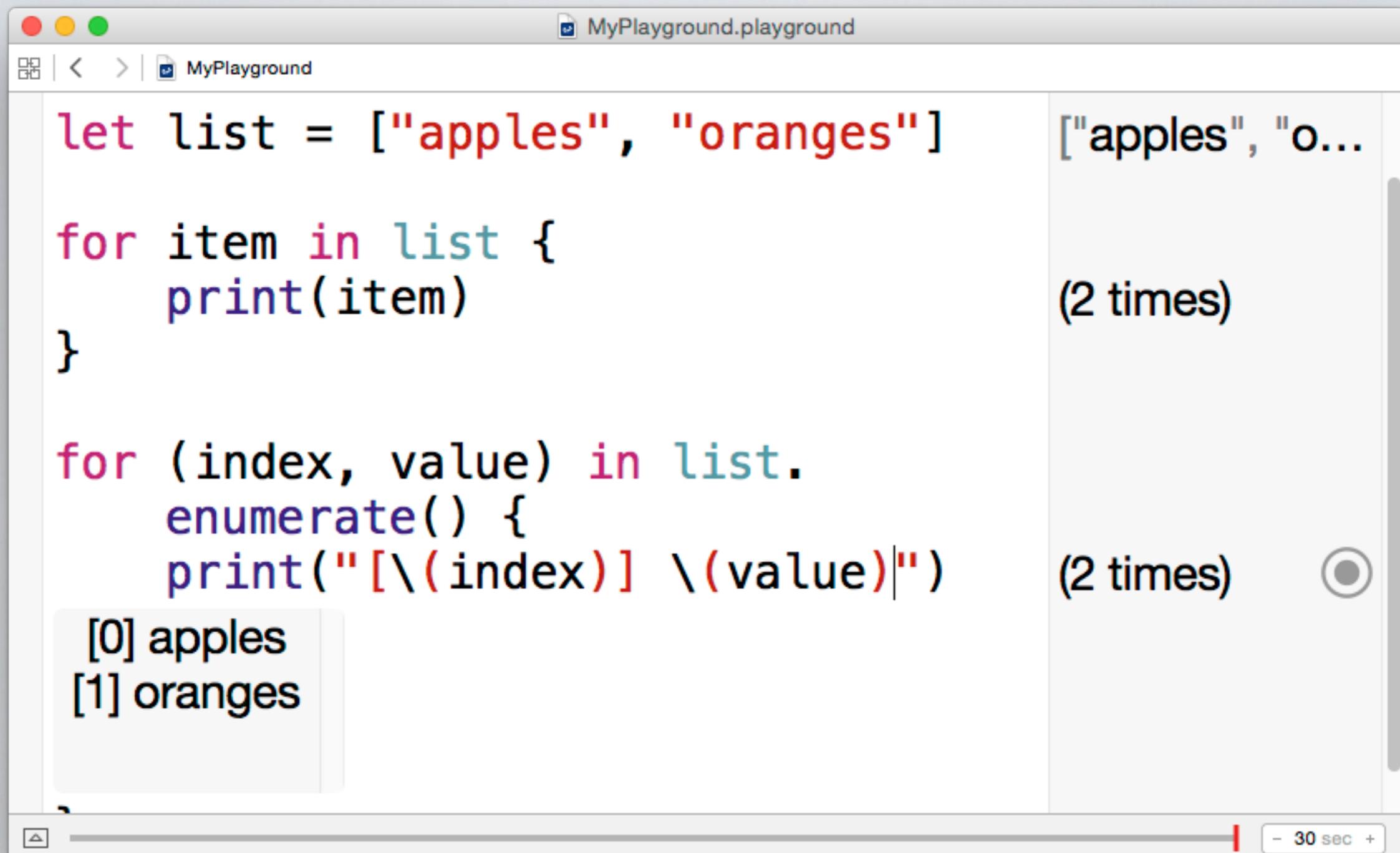
// Multi-dimensional array
let aArray = [[1,2], [3,4], [5,6]]
let bArray = [[Int]](count: 4,
    repeatedValue: [Int](count: 2,
    repeatedValue: 1))


```

A callout bubble on the right side of the screen highlights the line `repeatedValue: [Int](count: 2,` with a tooltip showing `[[1, 2], [3, 4], ...]`. Another part of the callout highlights the line `repeatedValue: 1))` with a tooltip showing `[[1, 1], [1, ...]]`.

The bottom status bar shows a timer at "30 sec".

COLLECTIONS



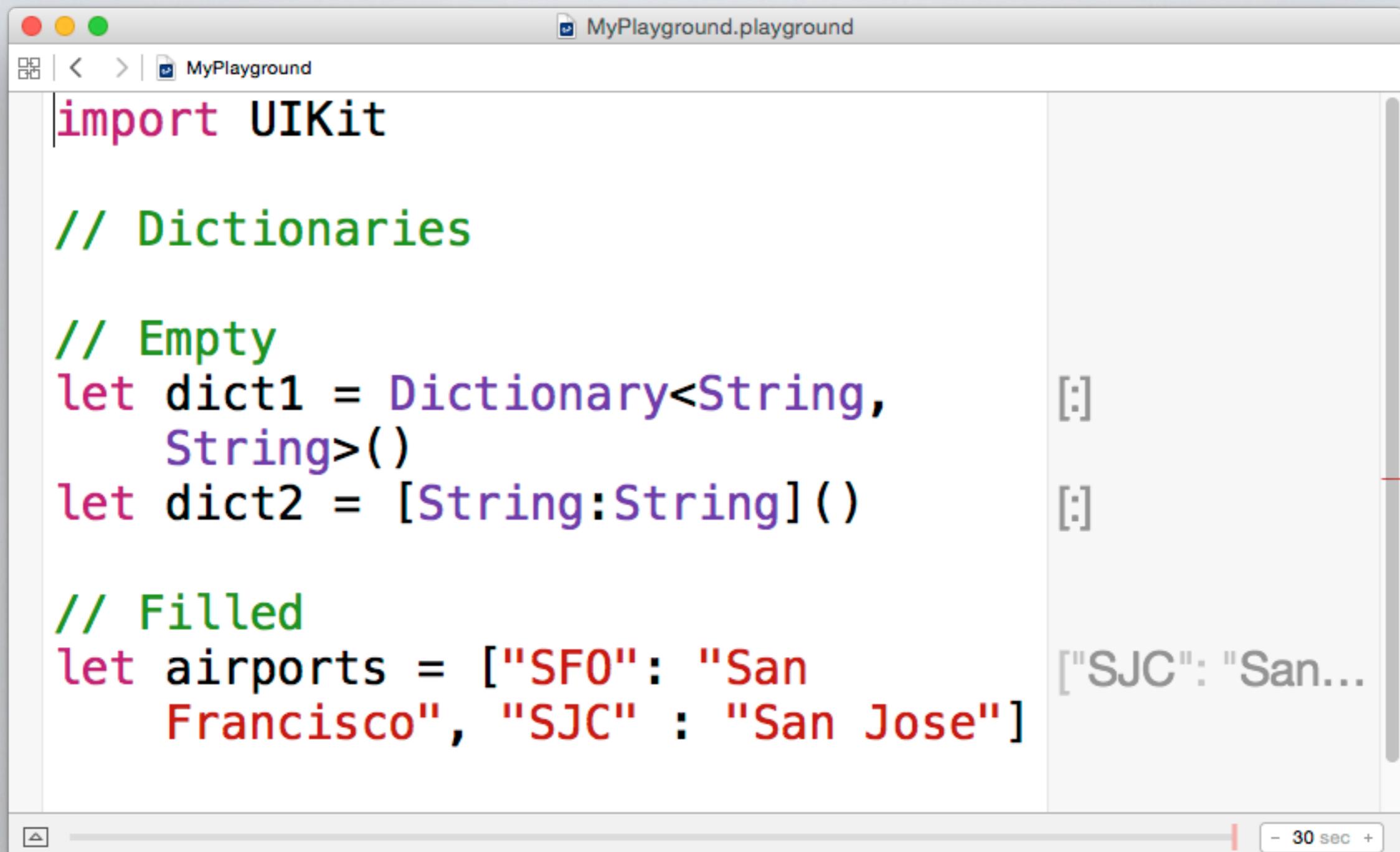
The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code in the playground creates an array "list" containing two strings: "apples" and "oranges". It then prints each item in the array using a standard for loop and an enumerate loop. The output on the right side of the playground shows the results of these prints.

```
let list = ["apples", "oranges"] ["apples", "o..."]

for item in list {
    print(item)
} (2 times)

for (index, value) in list.
    enumerate() {
        print("[\u{0028}index\u{0029] \u{0028}value\u{0029]")
}
[0] apples
[1] oranges
```

COLLECTIONS



The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains Swift playground code demonstrating various dictionary types:

```
import UIKit

// Dictionaries

// Empty
let dict1 = Dictionary<String, String>()
let dict2 = [String:String]()

// Filled
let airports = ["SF0": "San Francisco", "SJC" : "San Jose"]
```

The code defines two empty dictionaries: `dict1` using the generic type `Dictionary<String, String>()` and `dict2` using the array-like type `[String:String]()`. It then defines a filled dictionary `airports` containing key-value pairs for San Francisco and San Jose airports.

COLLECTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The playground contains the following Swift code:

```
// Filled
let airports = ["SF0": "San
Francisco", "SJC" : "San Jose"]

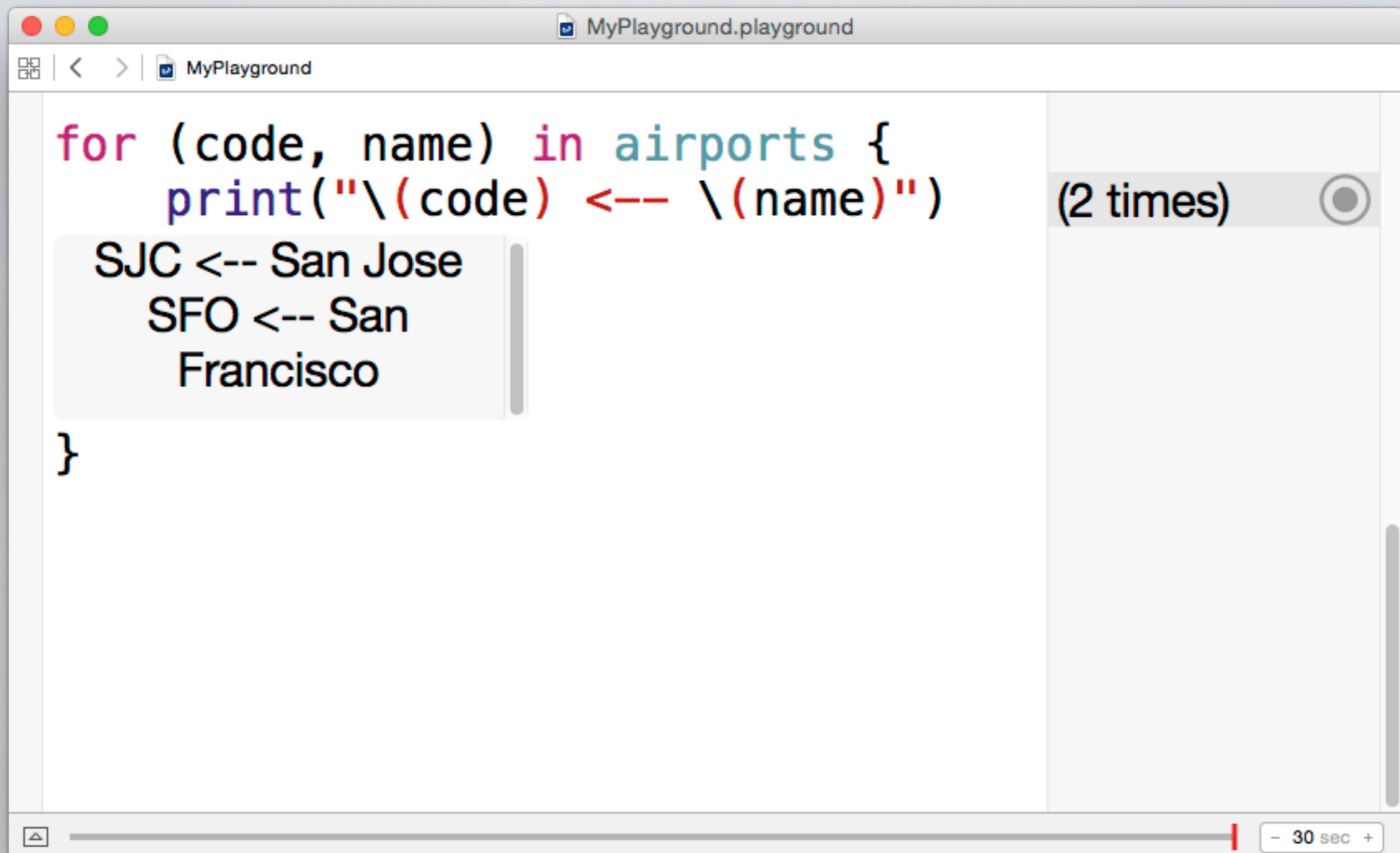
airports["SF0"]
airports["SJC"]
```

The output pane displays the results of the playground's execution:

```
["SJC": "San...
"San Francis...
"San Jose"
```

The bottom right corner of the window has a timer set to "30 sec".

COLLECTIONS



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground prints two entries from a collection named "airports".

```
for (code, name) in airports {
    print("\(code) <- \(name)")
}
SJC <- San Jose
SFO <- San
Francisco
```

The output pane shows the results of the print statements:

(2 times)

SJC <- San Jose
SFO <- San
Francisco

At the bottom of the playground window, there is a toolbar with a play button and a timer set to "30 sec".

COLLECTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". Inside, there are two code snippets in Swift:

```
for code in airports.keys {  
    print("\(code)")  
    SJC  
    SFO  
}  
  
for name in airports.values {  
    print("\(name)")  
    San Jose  
    San Francisco
```

Both snippets have a "(2 times)" output indicator and a circular disclosure button to the right.

COLLECTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The playground contains three code snippets demonstrating the use of Swift's Set collection.

```
// Empty set
var mySet = Set<Int>()
mySet.insert(1)
mySet.insert(2)
mySet.insert(1)
```

The output for the first snippet shows the state of the set after each insertion:

- Initial state: []
- After inserting 1: {1}
- After inserting 2: {2, 1}
- After inserting 1 again: {2, 1}

```
// From a list
var mySetFromList: Set<String> =
    ["a", "a", "b", "c"]
```

The output for the second snippet shows the final state of the set after inserting elements from a list:

- Final state: {"b", "a", "c"} (highlighted in gray)

```
// Provides usual set operations
// union, intersect, difference
```

The third snippet is partially visible at the bottom of the playground.

RANGES

The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

```
import UIKit

let closedRange = 1...5
for i in closedRange {
    print(i)
}
```

The output pane shows the results of the loop:

- "1..<6"
- (5 times)
-

The output is:
1
2
3
4
5

At the bottom of the playground window, there is a toolbar with a play button and a timer set to "30 sec".

RANGES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

```
import UIKit

let halfOpenRange = 1..<5
for i in halfOpenRange {
    print(i)
}
```

The output pane displays the results of the print statements:

```
1
2
3
4
```

Annotations on the right side of the code editor provide additional information:

- "1..<5" is annotated next to the range definition.
- "(4 times)" is annotated next to the loop iteration.
- A circular button icon is positioned next to the "(4 times)" annotation.

At the bottom of the playground window, there is a toolbar with a play button icon and a timer set to "- 30 sec +".

FUNCTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
import UIKit

// Function
// with no argument
// returns void
func myFunction() {
    print("myFunction")
}

// Call the function
myFunction()
```

The output pane on the right displays the result of running the code: "myFunction".

At the bottom of the window, there is a toolbar with a play button icon and a timer set to "- 30 sec +".

FUNCTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The playground contains the following Swift code:

```
import UIKit

// Function
// with no argument
// returns void (explicit)
func myFunction() -> Void {
    print("myFunction")
}

// Call the function
myFunction()
```

The output pane on the right shows the result of running the code: "myFunction".

FUNCTIONS

The screenshot shows a macOS application window titled "MyPlayground.playground". The main area contains Swift code:

```
// Function
// with one argument
// returns void (explicit)
func myFunction(myArg: Int) -> Void
{
    print("myFunction with \
        (myArg)")
}

myFunction with 1

// Call the function
myFunction(1)
```

The output pane on the right shows the result of running the code: "myFunction with 1". A tooltip "myFunct..." is visible over the output text.

FUNCTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
// Function
// with two arguments
// returns void (explicit)
func myFunction(myArg: Int,
                myOther: Int) -> Void {
    print("myFunction with \(myArg) " "myFunct...
        and \(myOther)")

    myFunction with 1 and 2
}

// Call the function
myFunction(1, myOther: 2)
```

A red box highlights the line "myFunction(1, myOther: 2)". In the output pane below, the text "myFunction with 1 and 2" is displayed. A small circular icon with a dot is visible next to the output text.

FUNCTIONS

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
import UIKit

// Function
// with two arguments
// returns Int
func myFunction(myArg: Int,
    myOther: Int) -> Int {
    return myArg + myOther
}

// Call the function
myFunction(1, myOther: 2)
```

On the right side of the code editor, there are two "3" icons indicating three test cases for the function. The bottom right corner of the window shows a timer set to "30 sec".

FUNCTIONS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code area contains a function definition and a call to it:

```
func sum(numbers: Int...) -> Int {
    var total = 0
    for number in numbers {
        total += number
    }
    return total
}

sum(1,2,3,4)
```

The right side of the playground shows the results of the function calls:

Call	Result
0	0
(4 times)	(4 times) 10
10	10
10	10

The bottom of the playground has a toolbar with a play button and a timer set to "30 sec".

FUNCTIONS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code area contains the following Swift code:

```
func sum(numbers: Int...) -> Int {
    var total = 0
    for number in numbers {
        Declaration let numbers: [Int]
        Declared In MyPlayground.playground
    }
    return total
}

// But you can't use an array
sum([1,2,3,4])
```

A callout box highlights the declaration of the variable "numbers" inside the loop. It shows the declaration "let numbers: [Int]" and the declaration location "Declared In MyPlayground.playground". To the right of the code, there are two output sections. The first section shows the result of the function call with an array of four integers, resulting in a total of 10. The second section shows an error for the commented-out line, indicating that arrays cannot be passed directly to variadic functions.

FUNCTIONS

- Other features not covered
 - default parameters
 - constant and variable parameters
 - inout parameters
 - throwing exceptions



OPTIONALS

A screenshot of an Xcode playground window titled "MyPlayground.playground". The code area contains:

```
import UIKit

var myValue: String?
myValue.dynamicType
```

The variable `myValue` is typed as `String?`. The expression `myValue.dynamicType` is highlighted in pink and shows a tooltip "Optional". To the right of the playground window, the inferred type is displayed as `nil` and `Optional<...>` with a circular disclosure button.

OPTIONALS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a function `misbehaving` that takes a `Bool` parameter and returns an `String!`. If the input is `false`, it returns the string "misbehaving"; if `true`, it returns `nil`. The playground then demonstrates two calls to `misbehaving`: one with `false` resulting in the string "misbehaving", and one with `true` resulting in `nil`. The bottom of the window shows a timeline with a red marker at 30 seconds.

```
func misbehaving(decision: Bool) -> String! {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

misbehaving(false).characters.count 11
misbehaving(true).characters.count
```

OPTIONALS

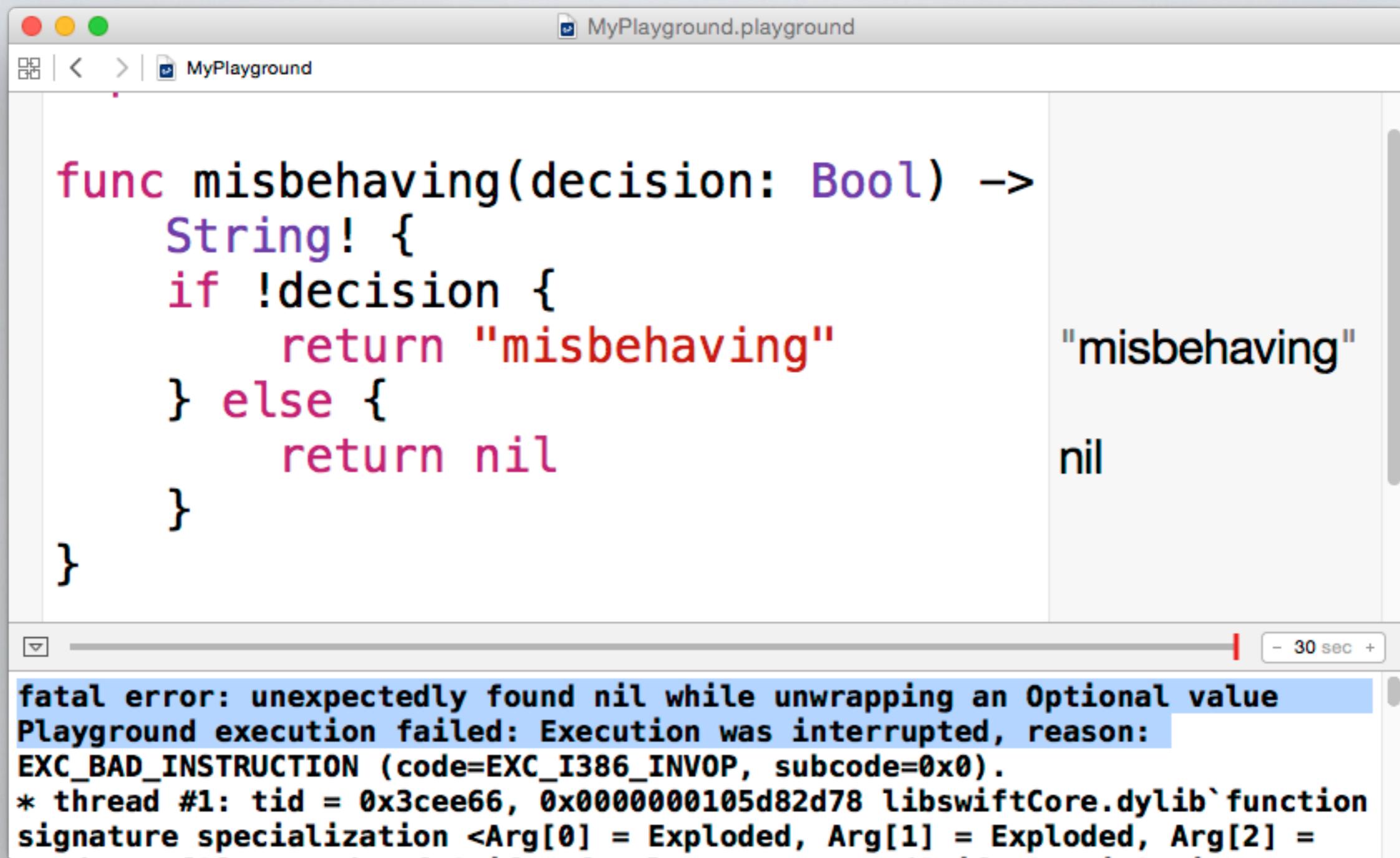
The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a function `misbehaving` that takes a `Bool` parameter and returns an `String!`. If the parameter is `false`, it returns the string "misbehaving"; if `true`, it returns `nil`. The playground then demonstrates two calls to `misbehaving`: one with `false` resulting in the string "misbehaving", and one with `true` resulting in `nil`. A red arrow points from the question "What happens here?" to the `nil` result.

```
func misbehaving(decision: Bool) -> String! {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

misbehaving(false).characters.count 11
misbehaving(true).characters.count
```

What happens here?

OPTIONALS



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground defines a function "misbehaving" that takes a Boolean parameter and returns an optional String. If the Boolean is false, it returns the string "misbehaving"; if true, it returns nil. The playground output shows the expected results: "misbehaving" when the input is false, and nil when the input is true. However, the error output at the bottom indicates a fatal error: "fatal error: unexpectedly found nil while unwrapping an Optional value". This error occurs because the code attempts to force-unwrap (using !) an optional value that contains nil, which leads to a bad instruction (EXC_BAD_INSTRUCTION).

```
func misbehaving(decision: Bool) -> String! {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

"misbehaving"
nil

fatal error: unexpectedly found nil while unwrapping an Optional value
Playground execution failed: Execution was interrupted, reason:
EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0).
* thread #1: tid = 0x3cee66, 0x0000000105d82d78 libswiftCore.dylib`function
signature specialization <Arg[0] = Exploded, Arg[1] = Exploded, Arg[2] =
```

OPTIONALS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a function "misbehaving" that returns an optional String. It checks if the input Bool is false; if so, it returns the string "misbehaving"; otherwise, it returns nil. A tooltip for the return value of "misbehaving(false)" indicates it's an optional type. Below the code, there are two error messages: one about unwrapping an optional type and another about using dot notation on an optional value. The bottom status bar shows a timer set to 30 seconds.

```
func misbehaving(decision: Bool) -> String? {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

misbehaving(false).characters.count
```

Value of optional type 'String?' not unwrapped; did you mean to use '!' or '??'

Value of optional type 'String?' not unwrapped; did you mean to use '!' or '??'

- 30 sec +

OPTIONALS

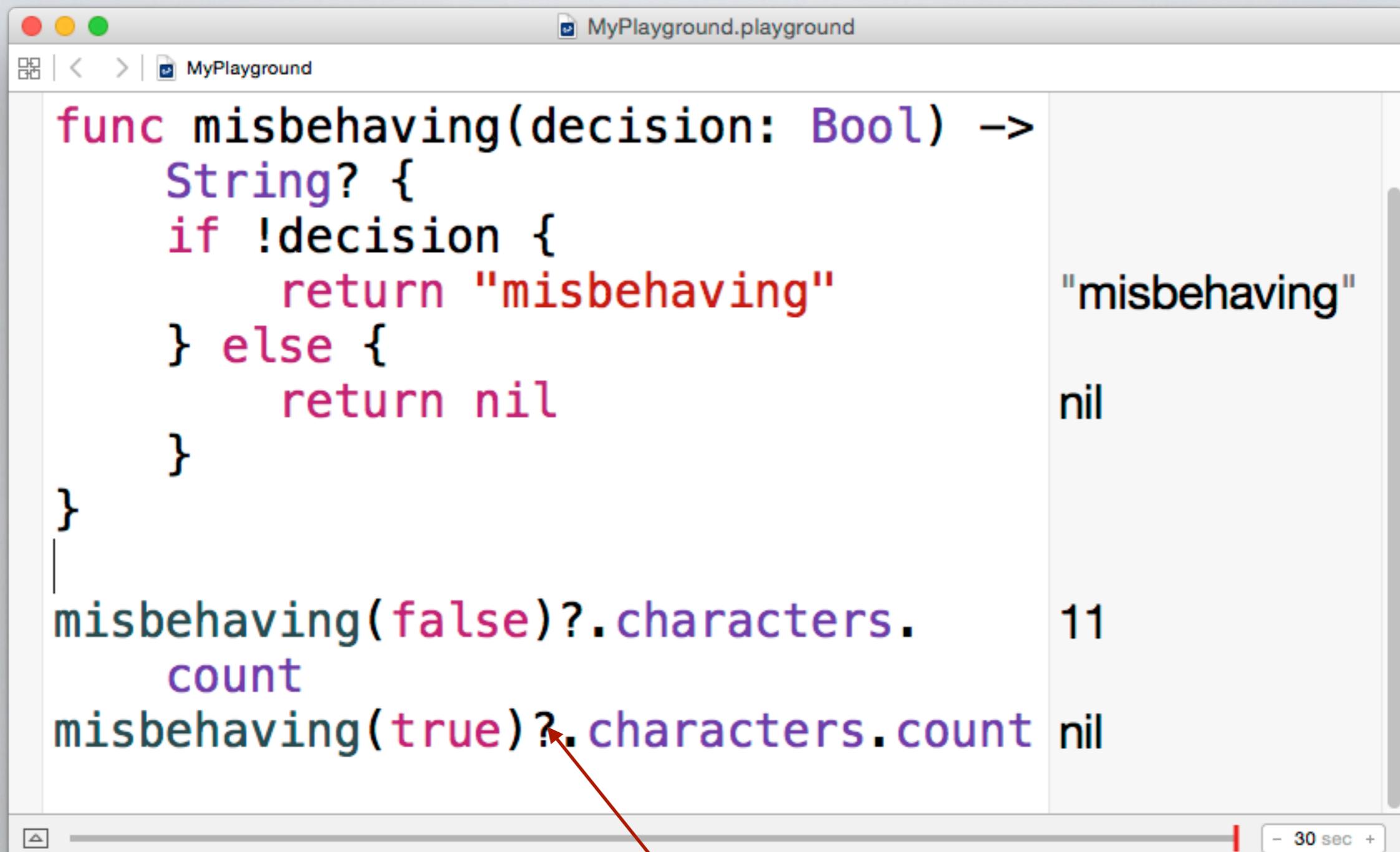
The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a function `misbehaving` that returns an optional string. If the input boolean is false, it returns the string "misbehaving"; if true, it returns nil. The playground then demonstrates two uses of this function: one with a forced unwrapping (using `!`) and another using nil coalescing (`?.`). The results are displayed on the right side of the playground.

```
func misbehaving(decision: Bool) -> String? {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

misbehaving(false)!.characters.count
misbehaving(true)?.characters.count
```

Result	Description
"misbehaving"	Return value when decision is false
nil	Return value when decision is true
11	Count of characters in "misbehaving"
nil	Nil coalescing result for true decision

OPTIONALS



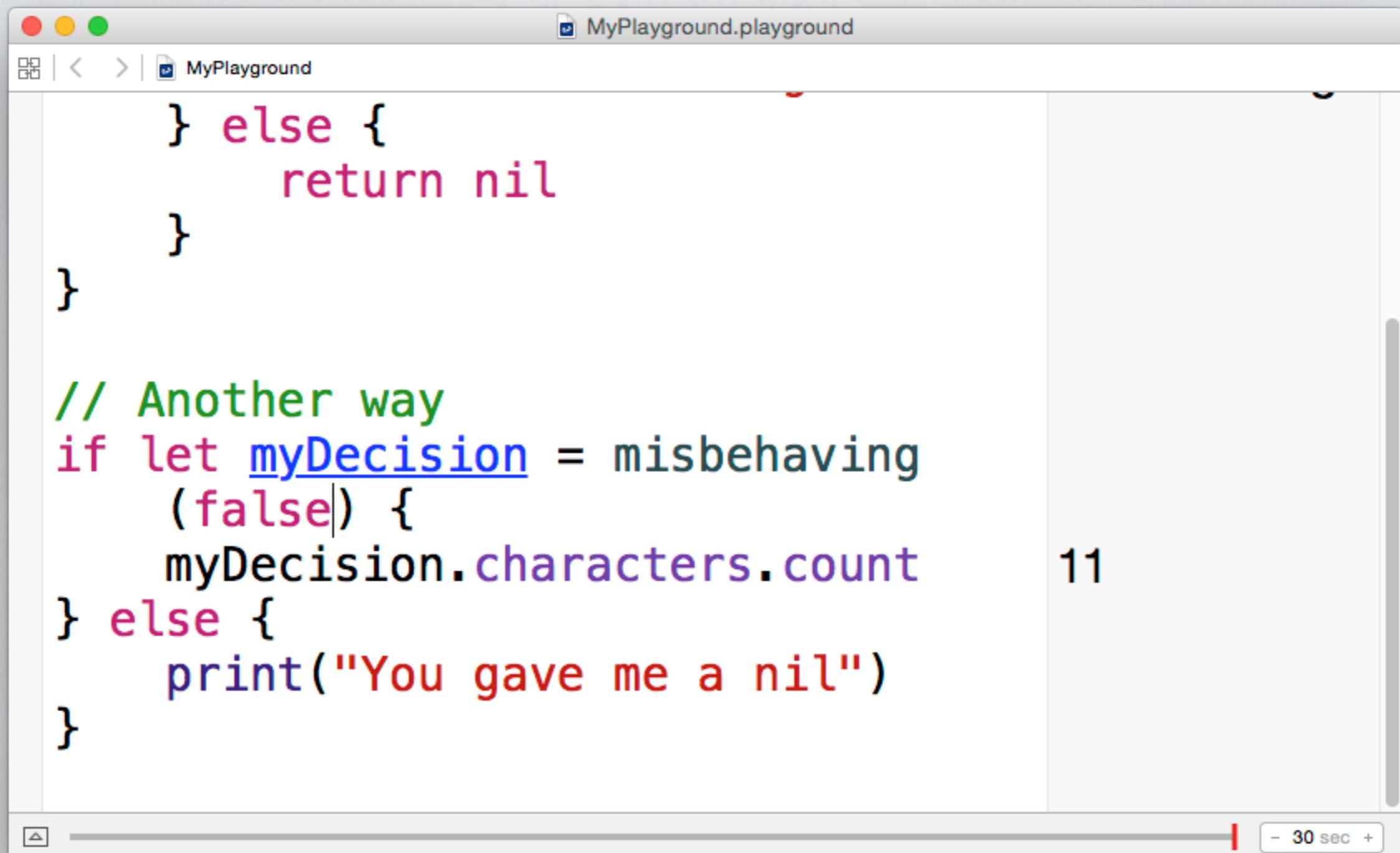
A screenshot of an Xcode playground window titled "MyPlayground.playground". The code defines a function "misbehaving" that returns an optional String. If the input Bool is false, it returns the string "misbehaving"; if true, it returns nil. The playground then demonstrates two calls to this function: one with a false argument resulting in a count of 11 characters, and another with a true argument resulting in nil. A red arrow points from the question mark in the optional binding "misbehaving(true)?" to the word "nil" in the output, highlighting the concept of nil coalescing.

```
func misbehaving(decision: Bool) -> String? {
    if !decision {
        return "misbehaving"
    } else {
        return nil
    }
}

misbehaving(false)?.characters.count
misbehaving(true)??.characters.count
```

If nil, don't call the rest of the methods after the '?'

OPTIONALS



The screenshot shows a Mac OS X window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
    } else {
        return nil
    }
}

// Another way
if let myDecision = misbehaving
    (false) {
    myDecision.characters.count
} else {
    print("You gave me a nil")
}
```

A vertical scroll bar is visible on the right side of the code editor. In the bottom right corner of the window, there is a timer set to "30 sec".

OPTIONALS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground handles nil values using optional binding:

```
    } else {
        return nil
    }
}

// Another way
if let myDecision = misbehaving
    (true) {
    myDecision.characters.count
} else {
    print("You gave me a nil")
```

The output pane shows the result of the execution:

You gave me a nil

nil

"You gav..."

The status bar at the bottom indicates a duration of "30 sec".

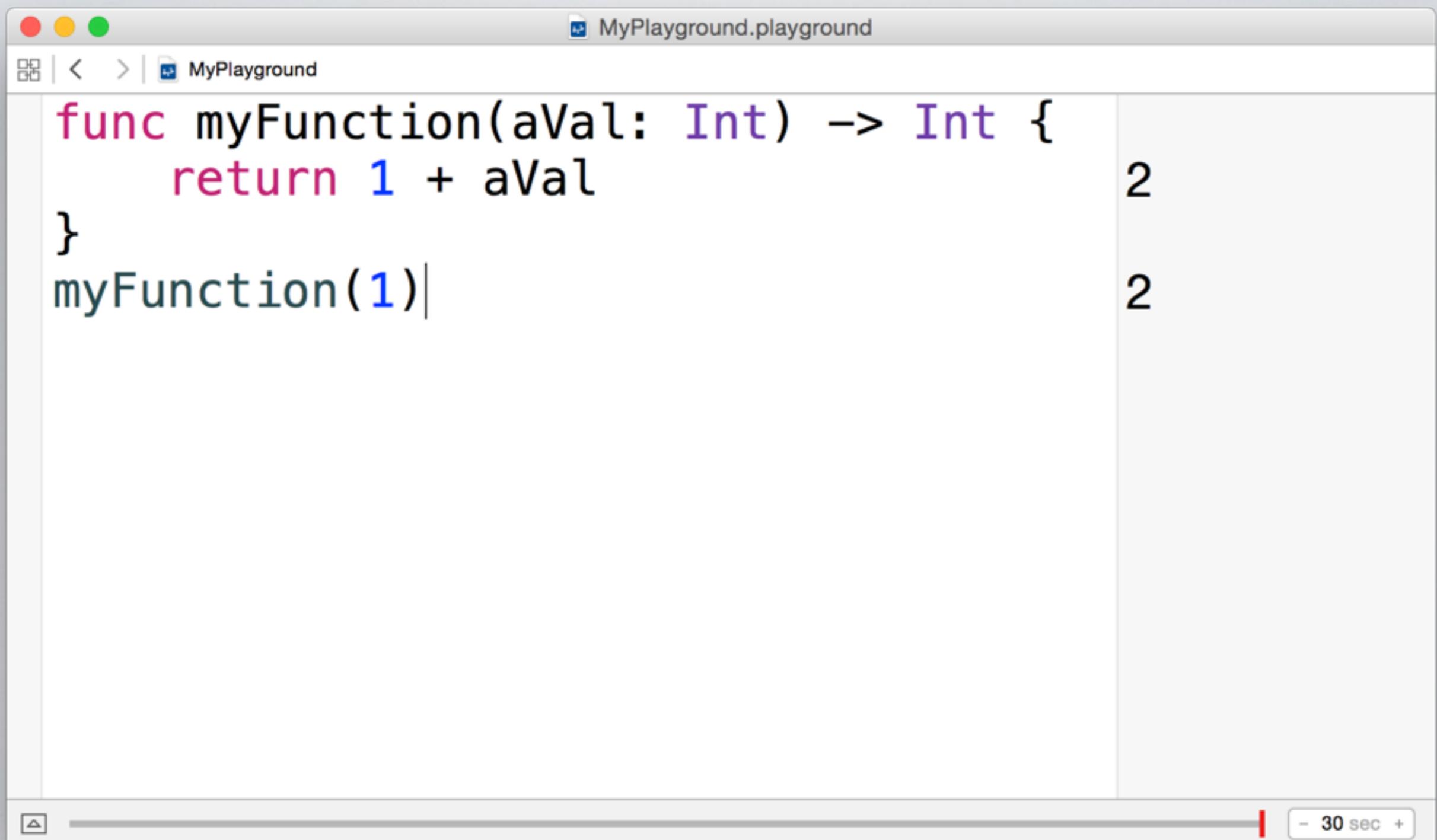
CLOSURES

Closures are **self-contained blocks** of functionality that can be passed around and used in your code.

Closures can **capture** and **store** references to any constants and variables from the context in which they are defined.



CLOSURES

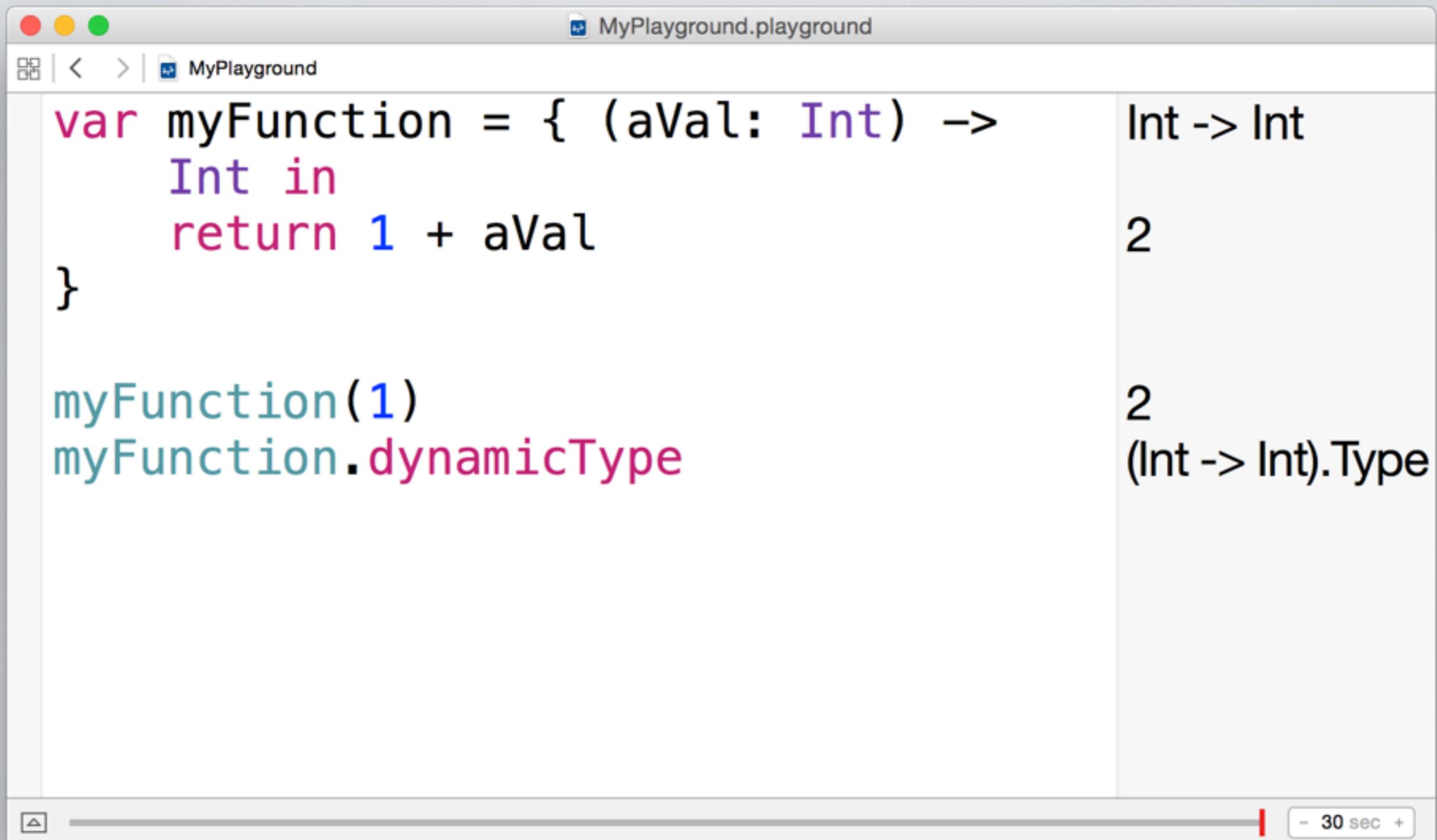


A screenshot of an Xcode playground window titled "MyPlayground.playground". The code area contains the following Swift code:

```
func myFunction(aVal: Int) -> Int {  
    return 1 + aVal  
}  
myFunction(1)
```

The output pane shows two results: "2" and "2", corresponding to the two invocations of the function. The bottom right corner of the output pane has a timer set to "- 30 sec +".

CLOSURES



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code area contains the following:

```
var myFunction = { (aVal: Int) ->
    Int in
    return 1 + aVal
}

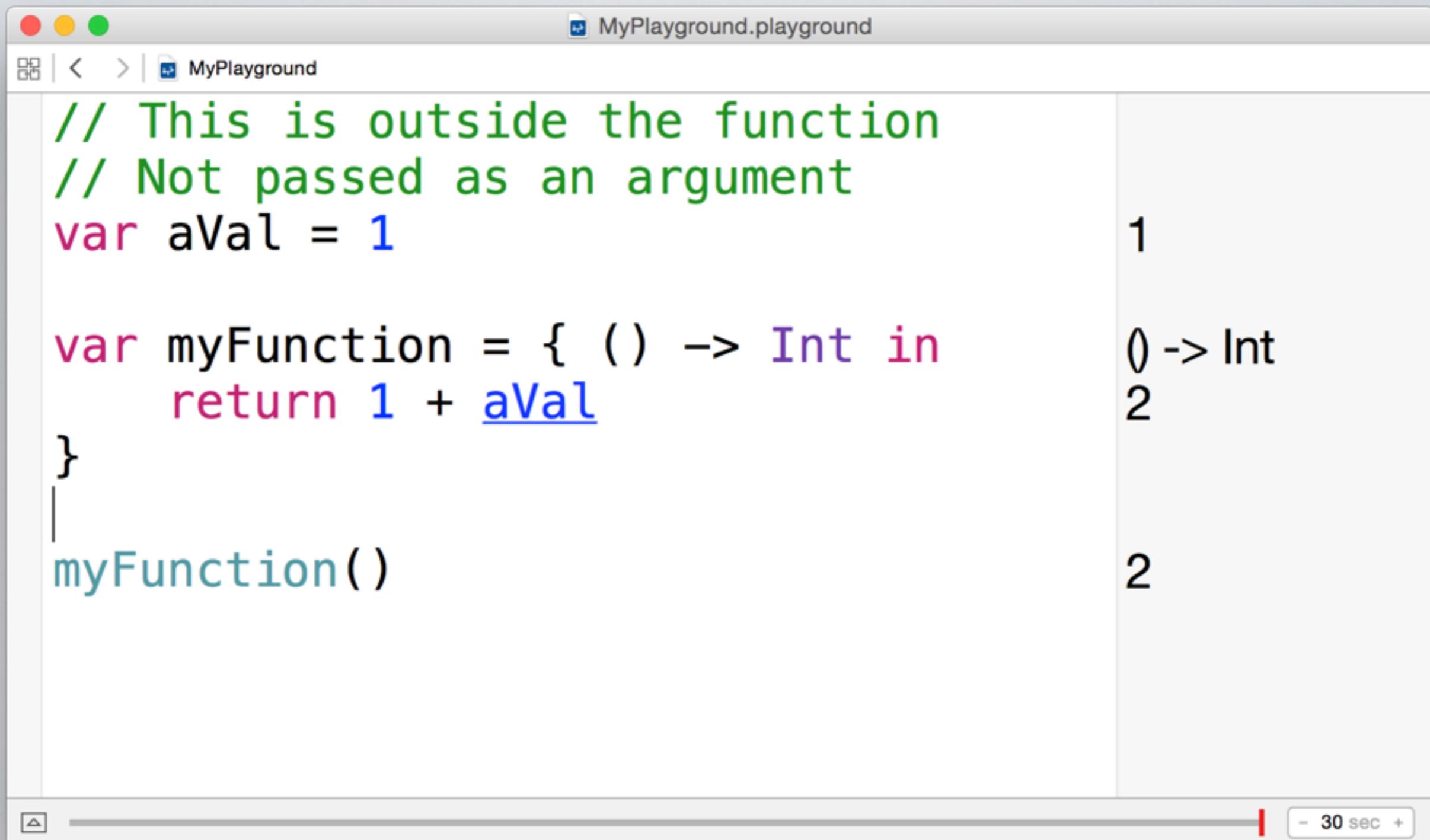
myFunction(1)
myFunction.dynamicType
```

The playground displays the results of the code execution:

- `Int -> Int`
- `2`
- `2`
- `(Int -> Int).Type`

At the bottom right of the playground window, there is a timer set to "30 sec".

CLOSURES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code demonstrates closures in Swift:

```
// This is outside the function
// Not passed as an argument
var aVal = 1

var myFunction = { () -> Int in
    return 1 + aVal
}

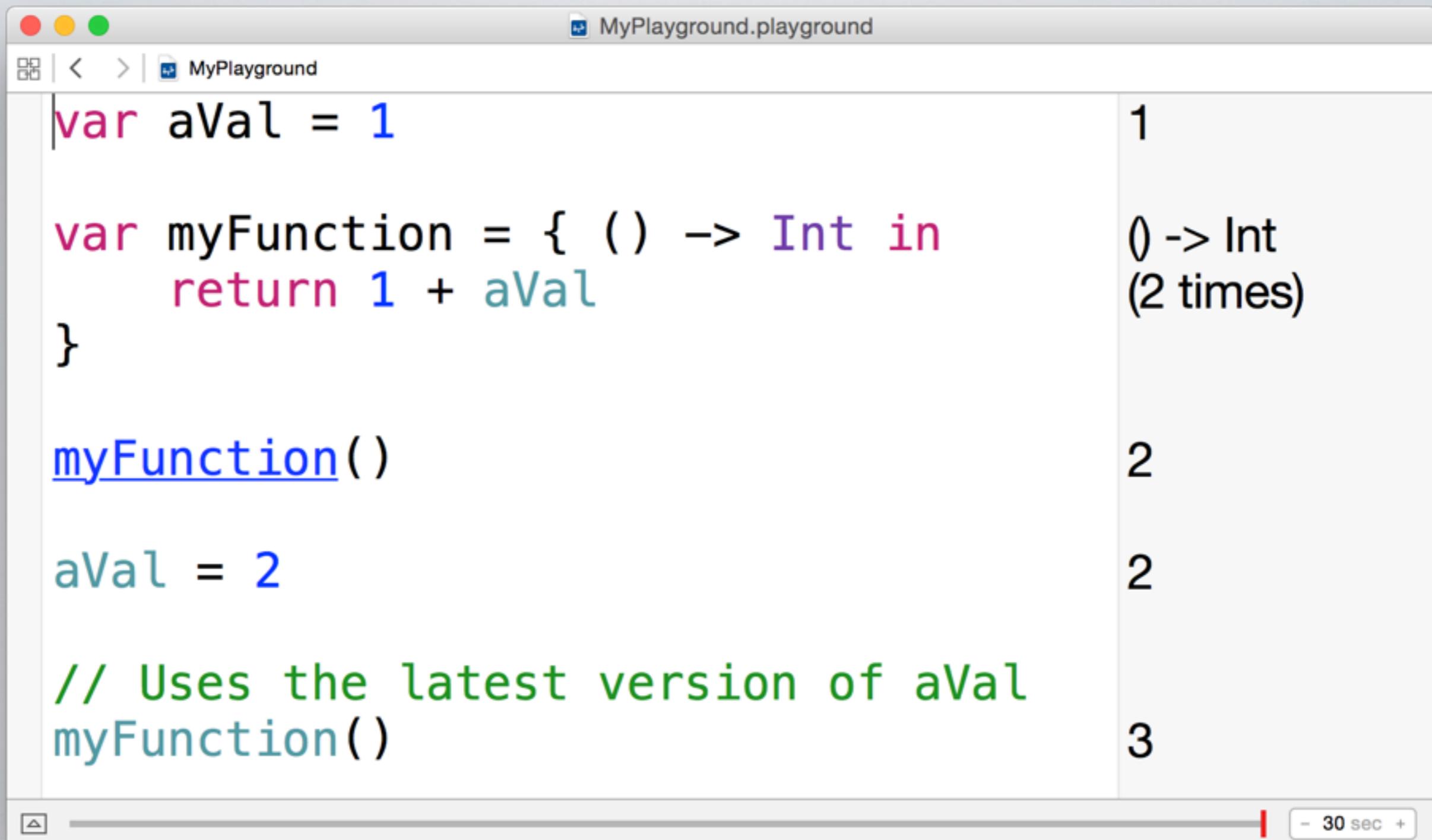
myFunction()
```

The output pane shows the results of the code execution:

Line	Value
1	1
0 -> Int	2
2	2

The bottom of the window shows a timer set to "30 sec".

CLOSURES



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code demonstrates closures capturing outer variable values.

```
var aVal = 1
var myFunction = { () -> Int in
    return 1 + aVal
}
myFunction()
aVal = 2
// Uses the latest version of aVal
myFunction()
```

The output pane shows the results:

Line	Value
1	1
2	0 -> Int (2 times)
3	2
4	2
5	3

The output shows that the closure captures the value of `aVal` at the time it was defined (1), even after the variable's value was changed to 2. The final output of `myFunction()` is 3, indicating the use of the latest value of `aVal`.

CLOSURES

SequenceType Protocol +

sequencetype

Pr SequenceType

Pr SequenceType

T SequenceType.Generator

f MusicSequenceGetSequence...

f MusicSequenceSetSequence...

f MusicSequenceGetSequence...

f MusicSequenceSetSequence...

E MusicSequenceType

T MusicSequenceType

T MusicSequenceType

M `contains(_:` Self.Generator.E...)

M `contains(_:` Self.Generator.El...)

M `elementsEqual(_:)`

M `elementsEqual(_:isEquivalent:)`

M `enumerate()`

M `filter(_:)`

M `flatMap<S : SequenceType>(_:_)`

M `flatMap<T>(_ : (Self.Generator...`

Inherits From
Not Applicable

Conforms To
Not Applicable

Nested Types
Not Applicable

Adopted By
[AnyBidirectionalCollection](#)
[AnyForwardCollection](#)
[AnyGenerator](#)
[AnyRandomAccessCollection](#)
[AnySequence](#) [Array](#) [ArraySlice](#)
[BidirectionalReverseView](#)
[CollectionOfOne](#) [ContinuousArray](#)

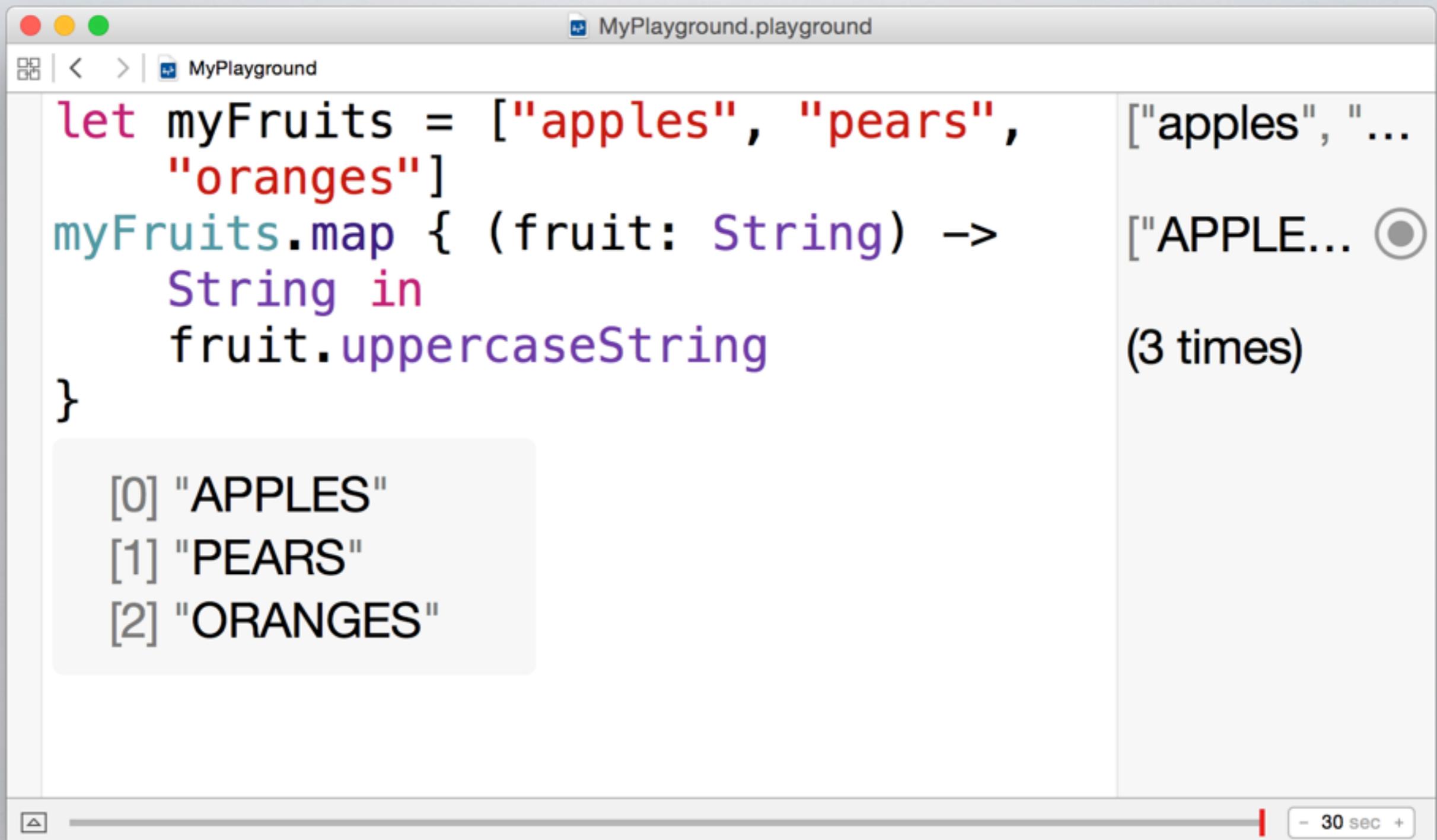
Import Statement
`import Swift`

Availability
Not Applicable

A type that can be iterated with a `for...in` loop.



CLOSURES

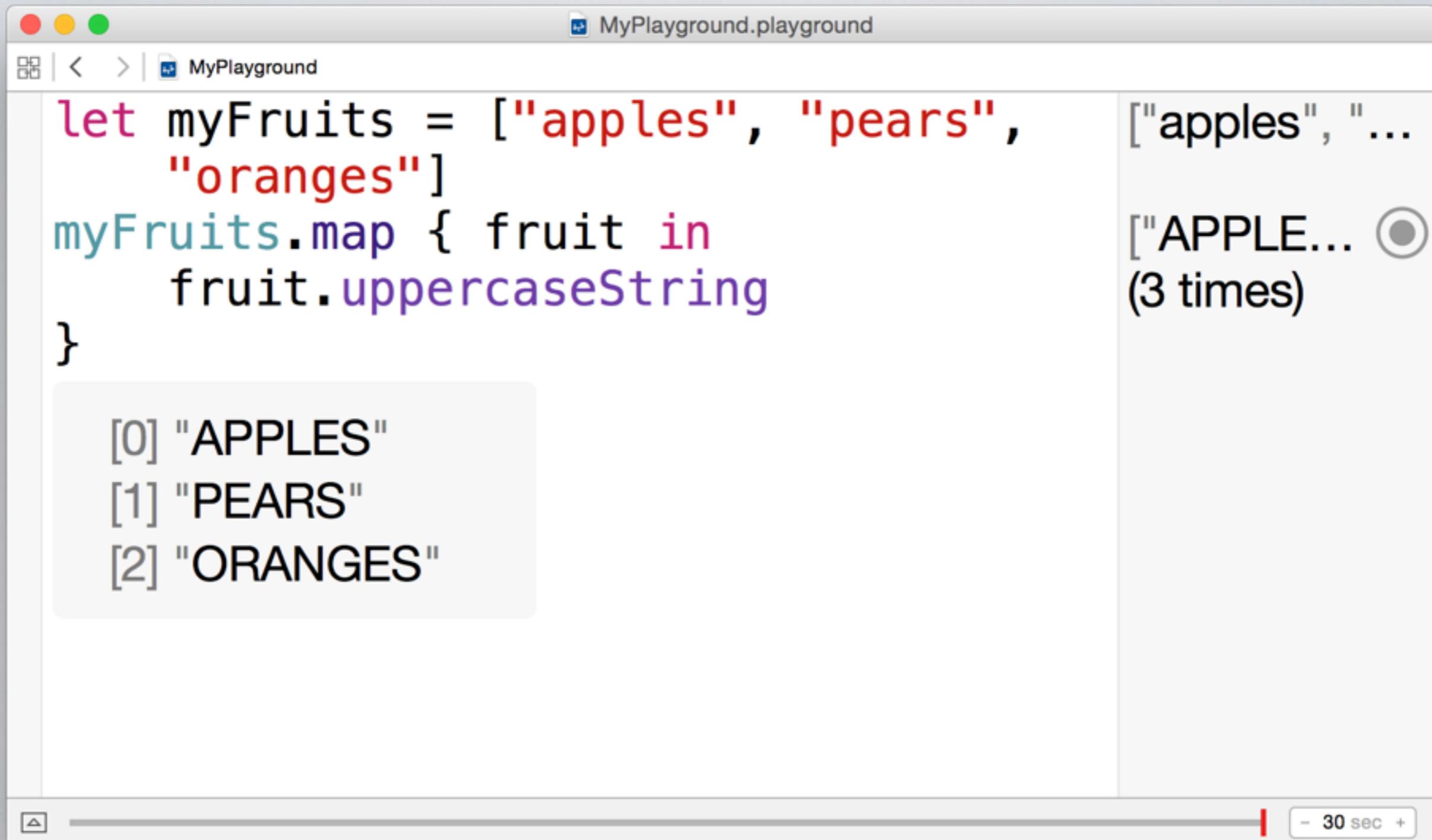


A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground is:

```
let myFruits = ["apples", "pears",  
               "oranges"]  
myFruits.map { (fruit: String) ->  
    String in  
    fruit.uppercaseString  
}  
  
[0] "APPLES"  
[1] "PEARS"  
[2] "ORANGES"
```

The output shows the result of the map operation: an array containing three uppercase strings: "APPLES", "PEARS", and "ORANGES". A tooltip for the first element, "[0] APPLES", is visible.

CLOSURES

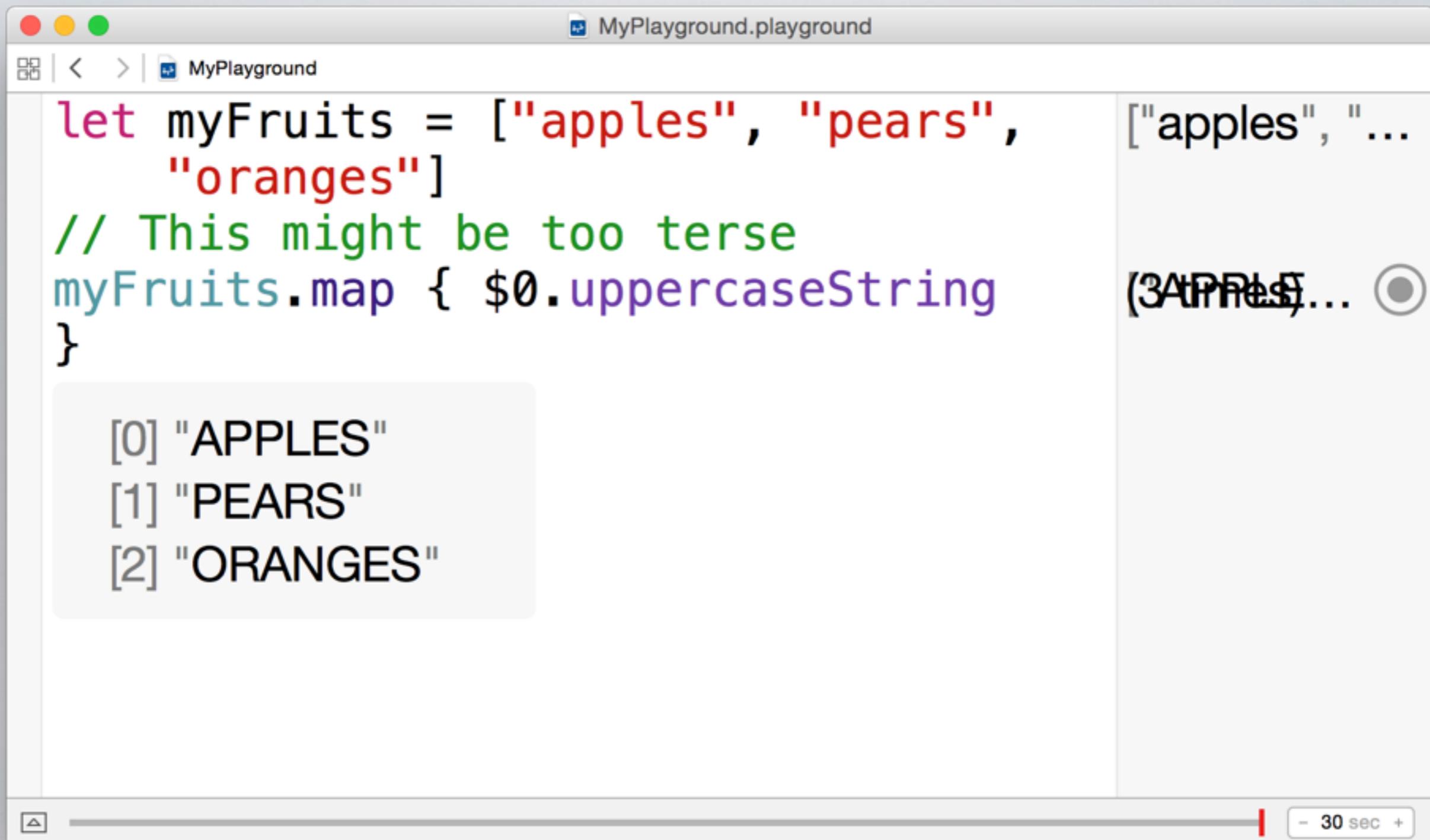


A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground is:

```
let myFruits = ["apples", "pears",  
               "oranges"]  
myFruits.map { fruit in  
    fruit.uppercaseString  
}  
  
[0] "APPLES"  
[1] "PEARS"  
[2] "ORANGES"
```

The output pane shows the result of the map operation: an array containing three uppercase strings: "APPLES", "PEARS", and "ORANGES". A tooltip for the first element, "[0] APPLES", is displayed.

CLOSURES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground is:

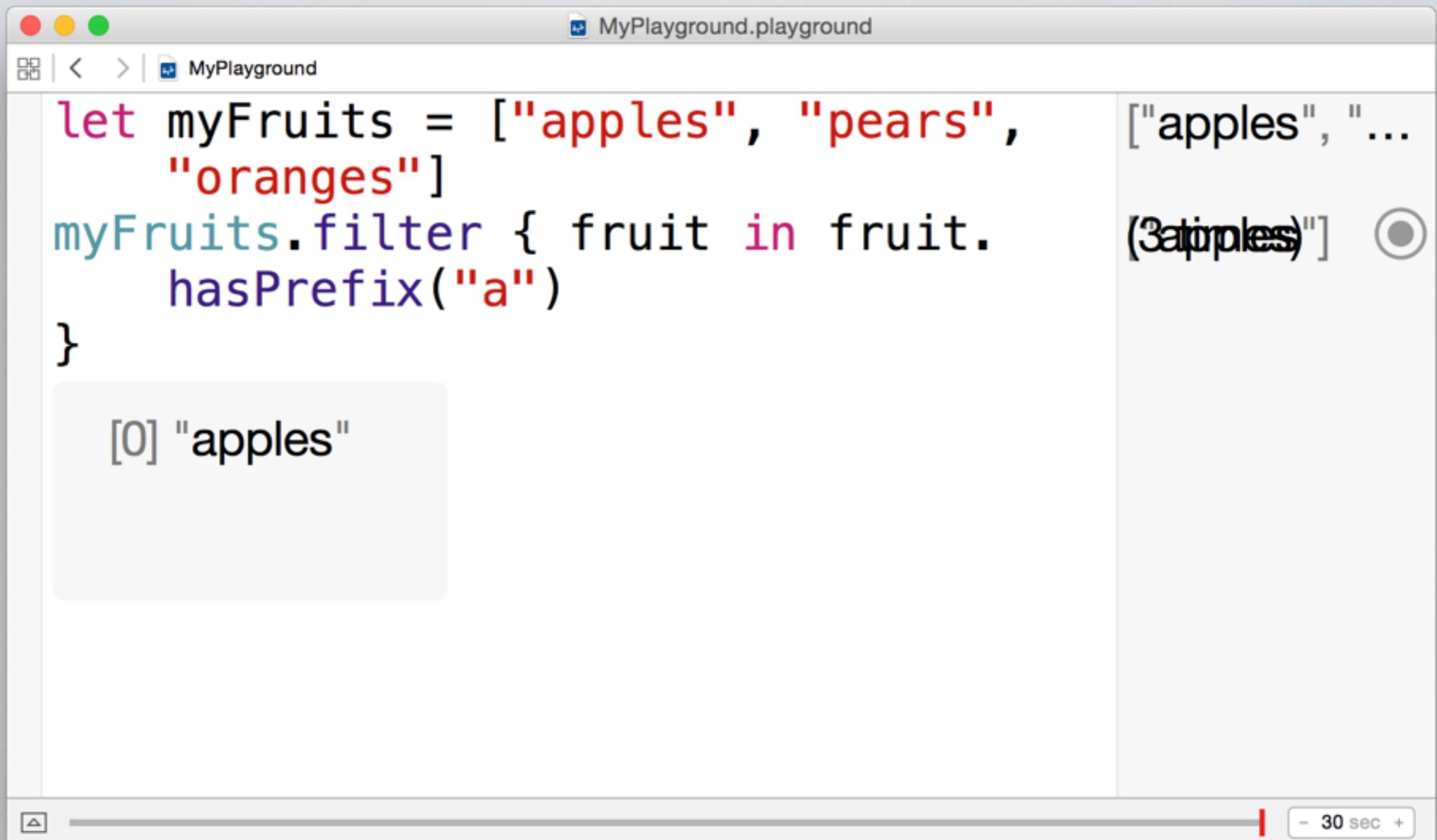
```
let myFruits = ["apples", "pears",  
               "oranges"]  
// This might be too terse  
myFruits.map { $0.uppercaseString  
}
```

The output pane shows the result of the map operation:

```
[0] "APPLES"  
[1] "PEARS"  
[2] "ORANGES"
```

The status bar at the bottom right indicates a timer of "30 sec".

CLOSURES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code in the playground is:

```
let myFruits = ["apples", "pears",  
               "oranges"]  
myFruits.filter { fruit in fruit.  
                  hasPrefix("a")  
}
```

The output pane shows the result of the filter operation:

```
[0] "apples"
```

The status bar at the bottom right indicates a timeout of "30 sec".

CLOSURES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is:

```
let myFruits = ["apples", "pears",  
               "oranges"]  
myFruits.sort { f1, f2 in f1 < f2  
}  
  
[0] "apples"  
[1] "oranges"  
[2] "pears"
```

The output pane displays the sorted array: ["apples", "oranges", "pears"]. A tooltip or callout box highlights the first element "[0] "apples"".

CLOSURES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The left pane contains Swift code that creates a container view, adds a red circle with a white center, and a white rectangle, then animates the circle's color and transform.

```
import UIKit
import XCPlayground

let containerView = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 375.0, height: 667.0))
XCPShowView("Container View", view: containerView)

let circle = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 50.0, height: 50.0))
circle.center = containerView.center
circle.layer.cornerRadius = 25.0

let startingColor = UIColor(red: (253.0/255.0), green: (159.0/255.0), blue: (47.0/255.0), alpha: 1.0)
circle.backgroundColor = startingColor

containerView.addSubview(circle)

let rectangle = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 50.0, height: 50.0))
rectangle.center = containerView.center
rectangle.layer.cornerRadius = 5.0

rectangle.backgroundColor = UIColor.whiteColor()

containerView.addSubview(rectangle)

UIView.animateWithDuration(2.0, animations: { () -> Void in
    let endingColor = UIColor(red: (255.0/255.0), green: (61.0/255.0), blue: (24.0/255.0), alpha: 1.0)
    circle.backgroundColor = endingColor

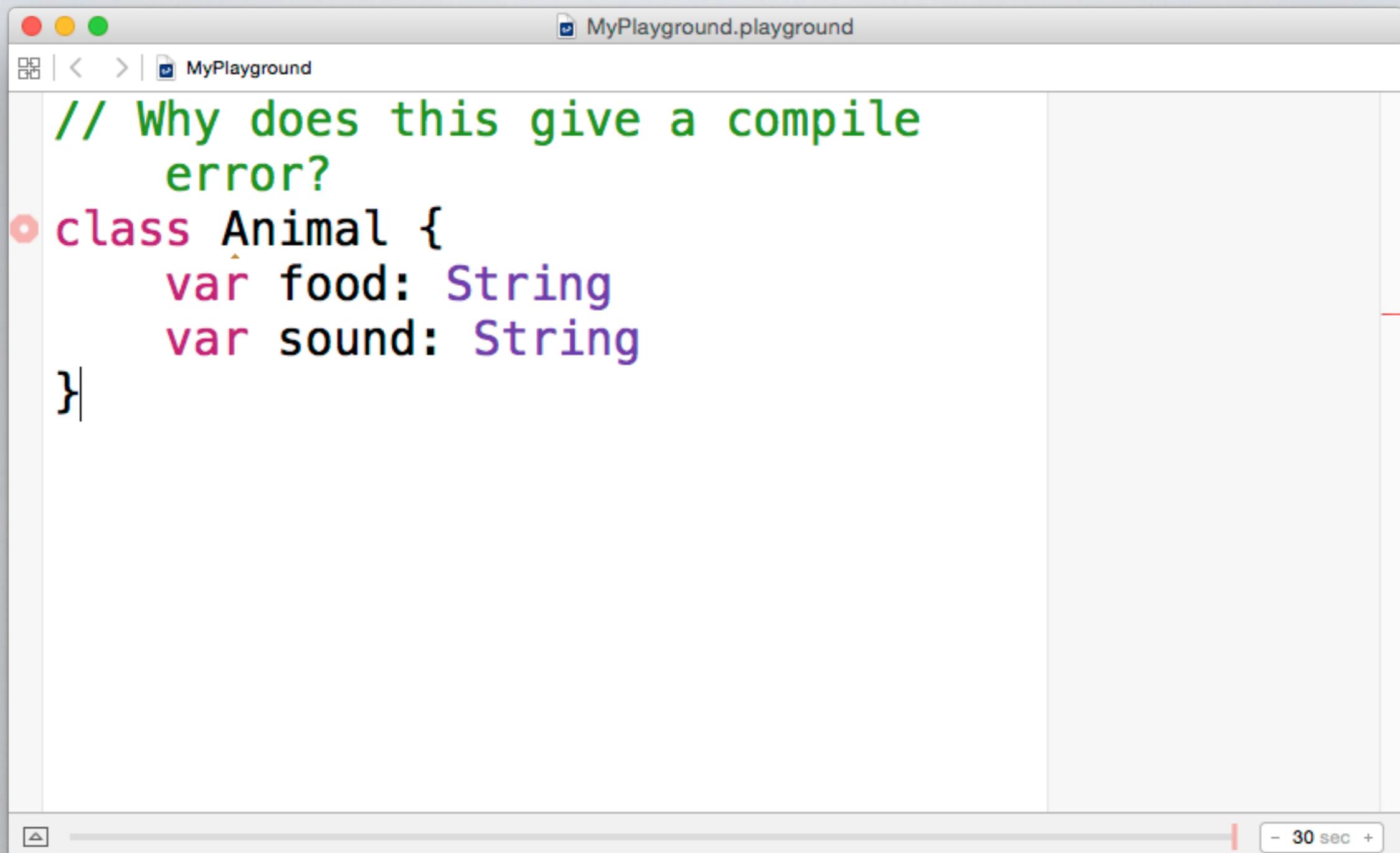
    let scaleTransform = CGAffineTransformMakeScale(5.0, 5.0)
    circle.transform = scaleTransform

    let rotationTransform = CGAffineTransformMakeRotation(3.14)

    rectangle.transform = rotationTransform
})
```

The right pane shows the resulting view hierarchy. It includes a "Container View" containing a "UIView" (the red circle) and a "UIView" (the white rectangle). The red circle has a CALayer with a radius of 0.992 and a color of orange-red. The white rectangle has a CALayer with a radius of 1.0 and a color of white. The bottom right corner of the slide shows a timeline slider at 30 seconds.

CLASSES



The image shows a screenshot of an Xcode playground window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
// Why does this give a compile  
error?  
class Animal {  
    var food: String  
    var sound: String  
}
```

A red circular icon with a white dot is positioned to the left of the word "class", indicating a syntax error. The status bar at the bottom right shows a timer set to "30 sec".

CLASSES



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code editor contains the following Swift code:

```
class Animal {
    var food: String
    var sound: String

    init(food: String, sound: String) {
        self.food = food;
        self.sound = sound;
    }
}

var cat = Animal(food: "fish", sound: "meow")
```

The word "Animal" is highlighted in blue, indicating it is a type. The bottom right corner of the window shows a timer at "30 sec".

CLASSES

```
MyPlayground.playground
MyPlayground

class Animal {
    var food: String
    var sound: String

    init(eats food: String,
         makesSound sound: String) {
        self.food = food;
        self.sound = sound;
    }
}

var cat = Animal(eats: "fish",
                 makesSound: "meow")
```

Animal

METHODS

The screenshot shows the Xcode documentation browser with the search bar containing "uitableviewda". The results list includes "UITableViewDataSource" and "UITableViewDelegate". The "UITableViewDataSource" entry is selected, showing its declaration and parameters.

Declaration

```
SWIFT  
func tableView(_ tableView: UITableView,  
 numberOfRowsInSection section: Int) -> Int
```

Parameters

<code>tableView</code>	The table-view object requesting this information.
<code>section</code>	An index number identifying a section in <code>tableView</code> .

Return Value

The number of rows in section.



CLASSES

The screenshot shows a Mac OS X window titled "MyPlayground.playground". The playground file is named "MyPlayground". The code in the playground is as follows:

```
class Animal {
    var food: String
    var sound: String

    init(eats food: String,
         makesSound sound: String) {
        self.food = food;
        self.sound = sound;
    }
}

var cat = Animal(eats: "fish",
                 makesSound: "meow")
```

A green callout bubble points from the bottom right towards the "Animal" variable in the code. The text inside the bubble reads: "Why doesn't this print the variables?".

STRING INTERPOLATION



The image shows a screenshot of Xcode's documentation interface. The search bar at the top left contains the text "customstringconvertible". The main content area is titled "CustomStringConvertible Protocol Reference".
Protocol Types: Not Applicable
Adopted By: NSObject, ObjCBool, Range, Selector, Set, StaticString, String, UTF16ViewString, String.UnicodeScalarView, UInt16, UInt32, UnicodeScalar, UnicodeScalarView
Availability: Not Applicable
Instance Properties:

- P description

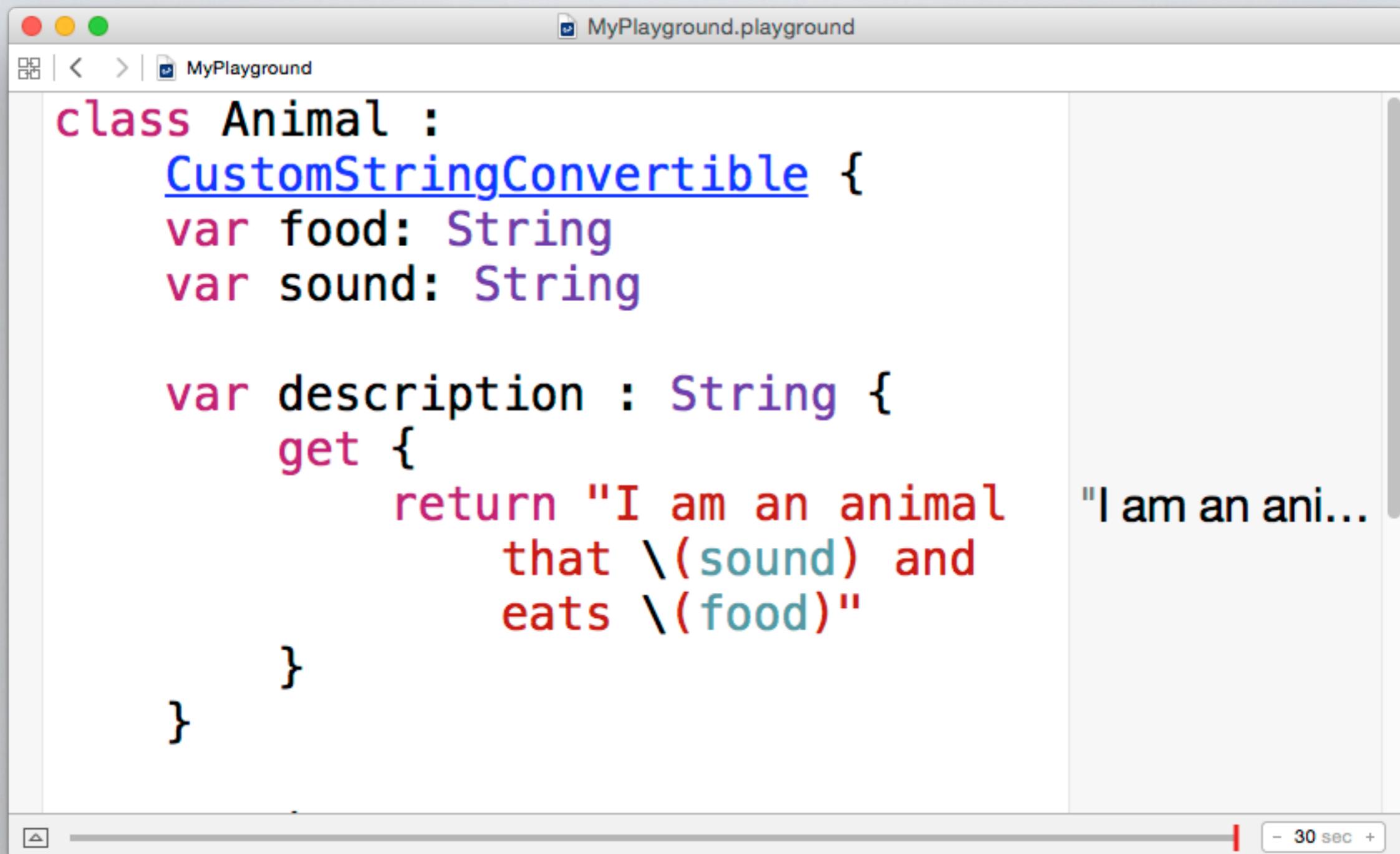
description Returns a textual representation.

The `description` representation is used when values are written to an output stream, for example, by `print`.

NOTE
String(instance) will work for an instance of *any* type, returning its description if the instance happens to be CustomStringConvertible. Using CustomStringConvertible as a generic constraint, or accessing a conforming type's description directly, is therefore discouraged.



PROTOCOLS



A screenshot of an Xcode playground window titled "MyPlayground.playground". The code defines a class named "Animal" that conforms to a protocol named "CustomStringConvertible". The class has two properties: "food" and "sound", both of type String. It also has a computed property "description" which returns a string describing the animal's sound and food. A tooltip for the "description" property shows the returned string: "I am an animal that \\$(sound) and eats \\$(food)". The playground interface includes standard Xcode controls like zoom and search at the bottom.

```
class Animal :  
    CustomStringConvertible {  
        var food: String  
        var sound: String  
  
        var description : String {  
            get {  
                return "I am an animal  
                    that \$(sound) and  
                    eats \$(food)"  
            }  
        }  
    }
```

PROTOCOLS

The screenshot shows the Xcode documentation browser with the search bar containing "customstringconvertible". The main content area displays the "CustomStringConvertible Protocol Reference". The protocol definition is as follows:

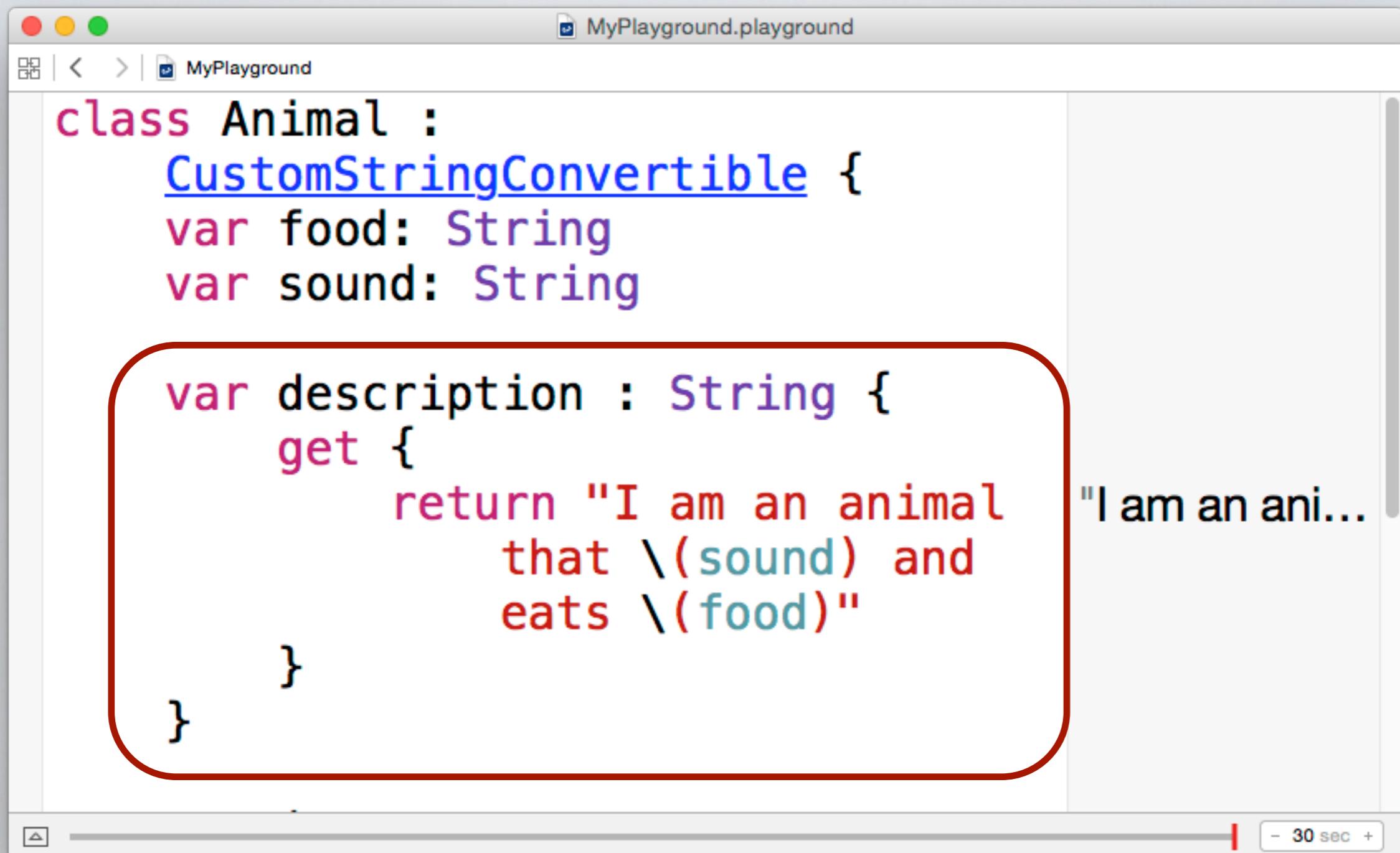
```
String(instance) will work for an instance of any type, returning its description if the instance happens to be CustomStringConvertible. Using CustomStringConvertible as a generic constraint, or accessing a conforming type's description directly, is therefore discouraged.
```

The "description" property is highlighted with a red rounded rectangle. Its declaration is shown as:

```
var description: String { get }
```

The sidebar on the left lists "Instance Properties" and "description". A purple "D" logo is visible in the bottom right corner.

PROTOCOLS



The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a class `Animal` that conforms to the protocol `CustomStringConvertible`. The class has properties `food` and `sound`, and a computed property `description` which returns a string describing the animal's sound and food.

```
class Animal :  
    CustomStringConvertible {  
        var food: String  
        var sound: String  
  
        var description : String {  
            get {  
                return "I am an animal  
                that \$(sound) and  
                eats \$(food)"  
            }  
        }  
    }  
}
```

A red rounded rectangle highlights the `description` property and its implementation. To the right of the highlighted code, the playground shows the result of evaluating the `description` property: "I am an ani...". At the bottom of the playground interface, there is a timer set to 30 seconds.

PROTOCOLS

A screenshot of an Xcode playground window titled "MyPlayground.playground". The code defines a protocol "CustomStringConvertible" and a class "Animal" that conforms to it.

```
class Animal :  
    CustomStringConvertible {  
        var food: String  
        var sound: String  
  
        var description : String {  
            get {  
                return "I am an animal  
                that \$(sound)s and  
                eats \$(food)"  
            }  
        }  
    }
```

The playground preview shows the result of the "description" property:

"I am an animal that meows and eats fish"

At the bottom right of the preview, there is a circular button with a dot in the center, indicating that the preview can be expanded.

PROTOCOLS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code is as follows:

```
MyPlayground.playground
MyPlayground

makesSound sound: String) {
    self.food = food;
    self.sound = sound;
}

var cat = Animal(eats: "fish",
    makesSound: "meow")
```

A red rounded rectangle highlights the text "food \"fish\"
sound \"meow\"". A green speech bubble points to this highlighted text with the question "Why didn't this change?". To the right of the variable declaration, there is a placeholder "I am an..." followed by a circular icon.

PROTOCOLS

The screenshot shows the Xcode interface with the title bar "CustomPlaygroundQuickLook" and a sidebar containing search results for "CustomPlaygroundQuickLook". The main content area displays the "CustomPlaygroundQuickLookable" protocol documentation.

CustomPlaygroundQuickLookable

protocol CustomPlaygroundQuickLookable

IMPORTANT

This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

Inherits From
Not Applicable

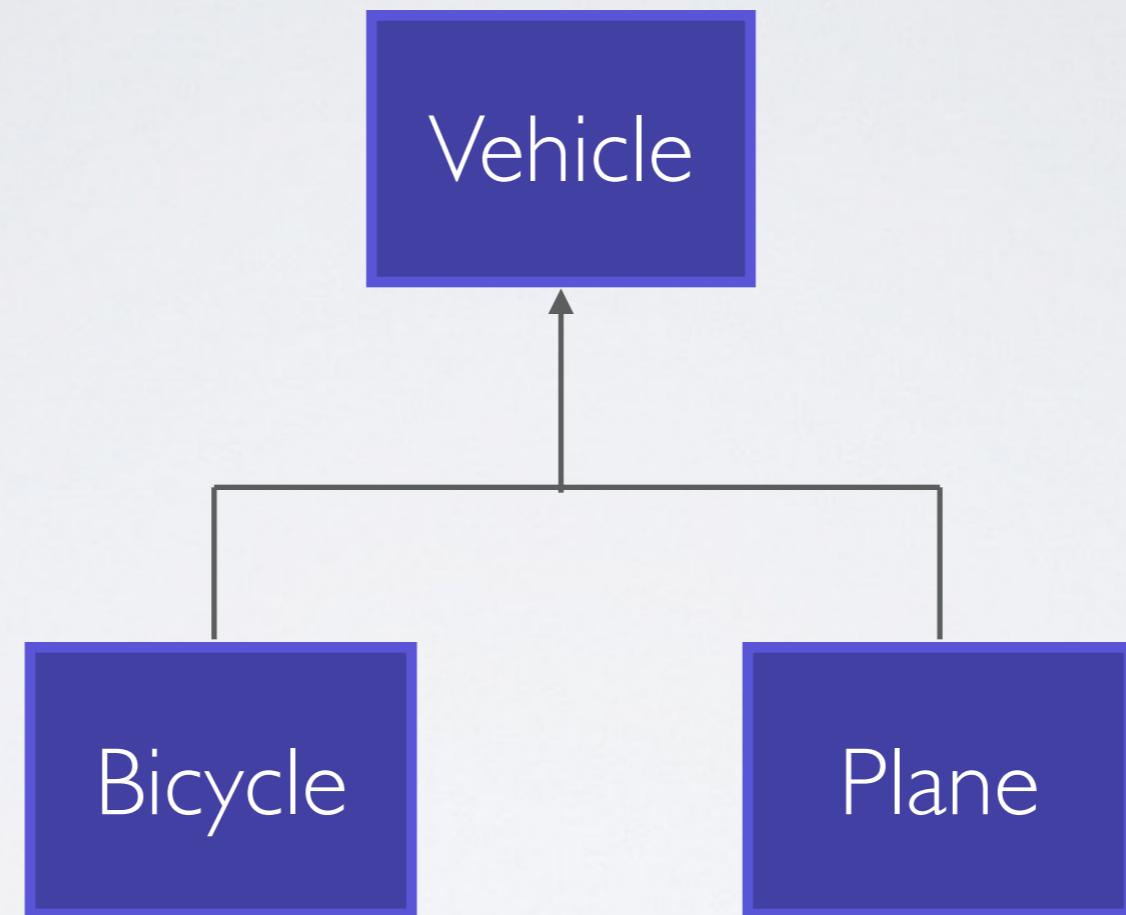
Conforms To
Not Applicable

Import Statement
`import Swift`

A green callout bubble with the text "Need to conform to this protocol. Try it!" is positioned above the protocol name.



CLASSES



CLASSES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a `Vehicle` class with a `currentSpeed` property and a `description` getter. An instance of `Vehicle` is created, and its `description` is printed to the output pane.

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "Moving at \
                (currentSpeed) miles per
                hour"
    }
}

var vehicle = Vehicle()
vehicle.description
```

The output pane displays the results:

- "Vehicle" (highlighted in the list)
- "Moving at..." (partially visible)

At the bottom right of the output pane, there is a timer set to "- 30 sec +".

CLASSES

The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code editor contains two class definitions:

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        currentSpeed = 5.0
    }
}

class Plane: Vehicle {
    override init() {
        super.init()
        currentSpeed = 500.0
    }
}
```

The code uses color-coded syntax highlighting: pink for keywords like `class`, `override`, and `super`; teal for types like `Vehicle`; and blue for variable names like `currentSpeed`.

CLASSES

The screenshot shows an Xcode playground window titled "MyPlayground.playground". It contains two code snippets demonstrating the use of the `description` property on classes.

```
var bicycleDesc = Bicycle().description
```

Moving at 5.0 miles per hour

```
var planeDesc = Plane().description
```

Moving at 500.0 miles per hour

The playground interface includes standard Xcode controls like file navigation and a search bar at the top, and a timeline at the bottom labeled "- 30 sec +".

STRUCTS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code defines a struct "Point" with two properties: "x" and "y", both initialized to 0. It then creates three instances: "point0" using the default initializer, "point1" using a constructor with arguments "x:1, y:1", and "point2" using a constructor with arguments "y:2, x:2". The last line is highlighted with a red error bar, indicating a syntax error: "Argument 'x' must precede argument 'y'".

```
struct Point {
    var x: Int = 0
    var y: Int = 0
}

var point0 = Point()
var point1 = Point(x:1, y:1)
// You must follow the property
// declaration order
! var point2 = Point(y:2, x:2)
    ! Argument 'x' must precede argument 'y'
```

STRUCTS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code demonstrates a struct named "Point" and its behavior:

```
var point1 = Point(x: 1, y:1)
var point2 = point1
point2.x = 5 // doesn't change the
              original point1
print(point1)
```

The output of the first print statement is "Point(x: 1, y: 1)".

```
print(point2)
```

The output of the second print statement is "Point(x: 5, y: 1)".

Annotations on the right side of the code highlight the variables and their types:

- point1: Point
- point2: Point
- point2.x: Point
- print(point1): "Point(x:..." (with a disclosure triangle)
- print(point2): "Point(x:..." (with a disclosure triangle)

At the bottom of the playground window, there is a timer set to "- 30 sec +".

STRUCTS

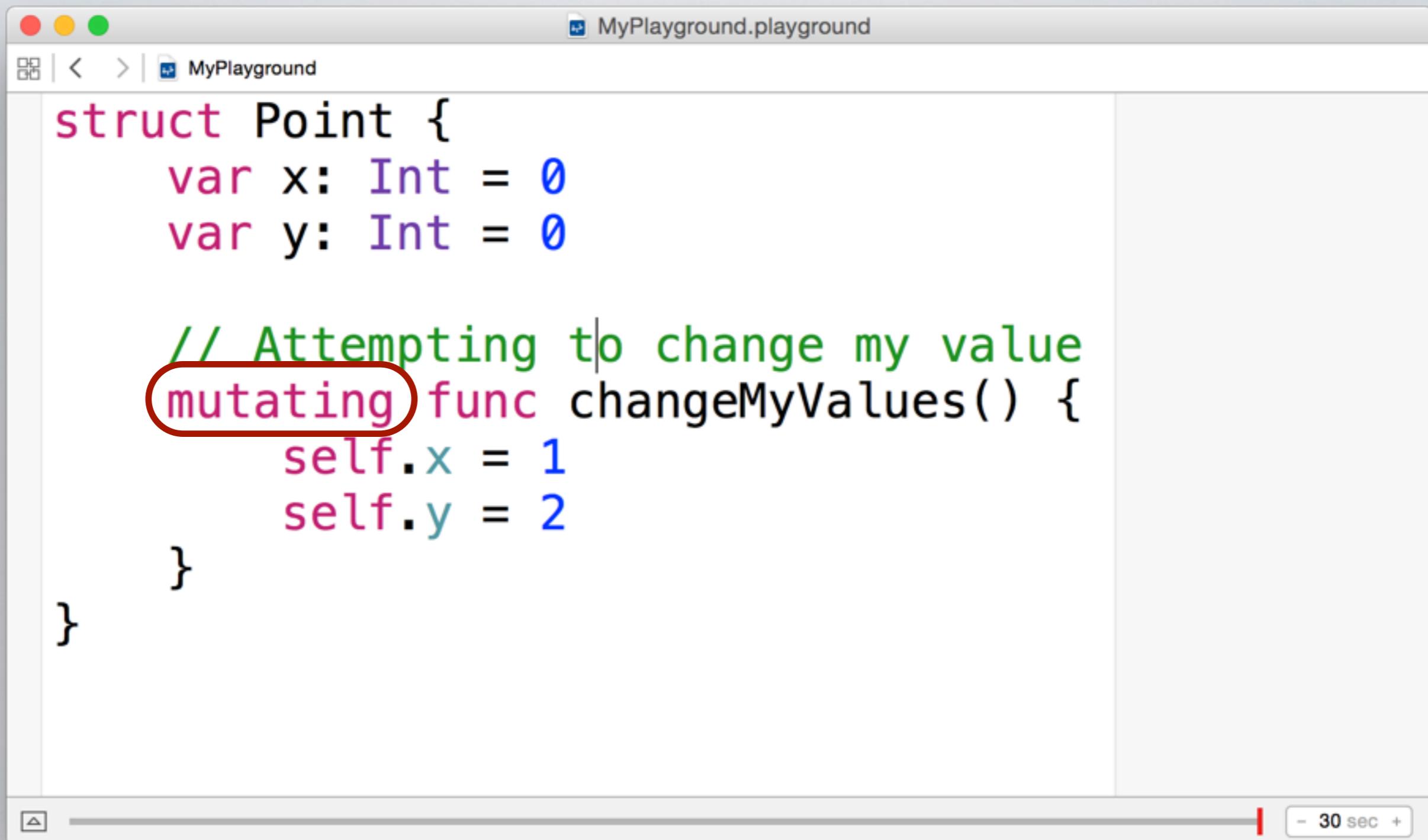
The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code is as follows:

```
struct Point {
    var x: Int = 0
    var y: Int = 0

    // Attempting to change my value
    func changeMyValues() {
        self.x = 1 // Cannot assign to prop...
        self.y = 2
    }
}
```

The line `self.x = 1` is highlighted with a red background and has a red error icon next to it, indicating a syntax error. A tooltip "Cannot assign to prop..." is visible over the error icon. The rest of the code is displayed in standard Xcode syntax highlighting.

STRUCTS



```
MyPlayground.playground
MyPlayground

struct Point {
    var x: Int = 0
    var y: Int = 0

    // Attempting to change my value
    mutating func changeMyValues() {
        self.x = 1
        self.y = 2
    }
}
```

CLASSES VS STRUCTS

- Structs for small-ish types where values are more important than identity: Currency, Point, Dates, etc.
- Structs for cases where you don't want to share data (concurrency)
- Structs if you think in functional programming

ENUMS

```
MyPlayground.playground
MyPlayground

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

var directionToHead = CompassPoint.South
```



The Swift
Programming
Language



ENUMS

```
MyPlayground.playground
MyPlayground

switch directionToHead {
    case .North:
        print("Lots of planets have a
              north")
    case .South:
        print("Watch out for penguins")    "Watch out...
    case .East:
        print("Where the sun rises")
    case .West:
        print("Where the skies are
              blue")
}
```



WHEN TO USE ENUMS?

- When you have a *collection* of *data* that belong together
- When the collection of data is *bounded*, e.g., enum Directions { North, South, East, West}
- If you want to get creative, pattern matching and functional programming

EXTENSIONS

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code



EXTENSIONS

The screenshot shows an Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

```
// Adds a new method to Int
extension Int {
    func repetitions(task: (val: Int) -> Void) {
        for i in 0..

The output of the playground shows the number 4 followed by four instances of the string "(4 times)", indicating that the extension's repetitions method was called with a closure that prints the value of i.


```

NEXT TIME

- Applying what we have learned to make HTTP requests
- Skim through AlamoFire [https://github.com/
Alamofire/Alamofire/tree/swift-2.0/Source](https://github.com/Alamofire/Alamofire/tree/swift-2.0/Source) code to test our knowledge of Swift