Introducing Swift

iOS Praktikum WS15/16

Session 01

“Tell me and I will forget.
Show me and I will remember.
Involve me and I will understand.
Step back and I will act.”

Copyright, Content & Referrals and Links

Technische Universität München

Lehrstuhl für Angewandte Softwaretechnik

-
- Content
  - TUM LS1 reserves the right not to be responsible for the topicality, correctness, completeness or quality of the information provided.
  - Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected.
  - All offers are not-binding and without obligation. Parts of the pages or the complete publication including all offers and information might be extended, changed or partly or completely deleted by the author without separate announcement.
- Referrals and Links
  - The author is not responsible for any contents linked or referred to from this document.
  - If any damage occurs by the use of information presented there, only the author of the respective pages might be liable, not the one who has linked to these pages.
  - Furthermore the author is not liable for any postings or messages published by users of discussion boards, guestbooks or mailing lists provided on his page.

➡ By obtaining and reading this document, you agree to this copyright statement.

Outline

- History and Introduction
- Swift - The Basics
  - Variables and Constants
  - Fundamental Types
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Outline

- **History and Introduction**
- Swift - The Basics
  - Variables and Constants
  - Fundamental Types
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

History and Introduction

- New - Introduced by Apple at WWDC 2014
- Object-oriented programming language
- Successor of Objective-C
- The language of your choice for building iOS and OS X applications
- LLVM Compiler is able to compile Swift, Objective-C , C and C++

Design goals

- Modern
- Safe by default
- Design for generality
- Fast and powerful
- Should feel like a scripting language
- Unification
- Combination of object oriented and functional concepts

New and advanced language concepts

Optionals

Protocol extensions

Type inference

Tuples

Lazy variables

Value semantics

Unicode support

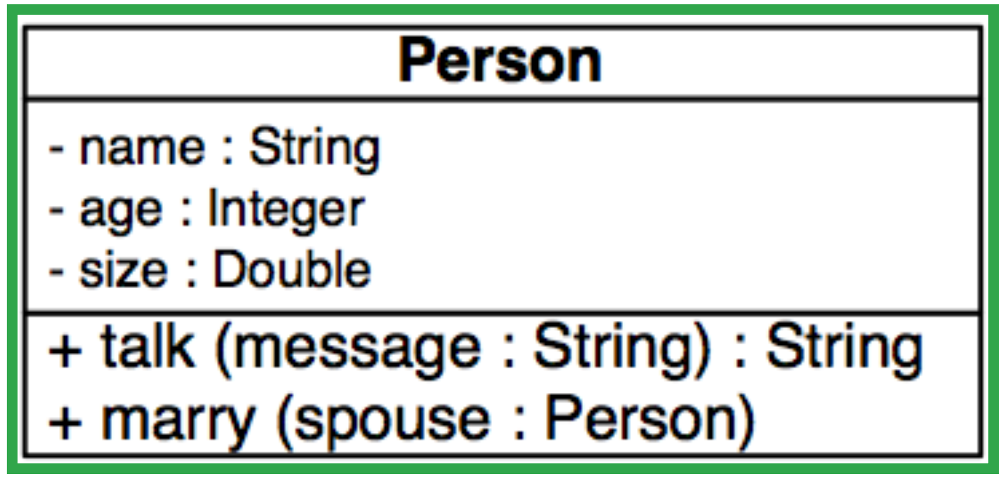
Generics

Namespaces

Closures

OOP Terminology in Swift (1)

- The basic concept is that of a class



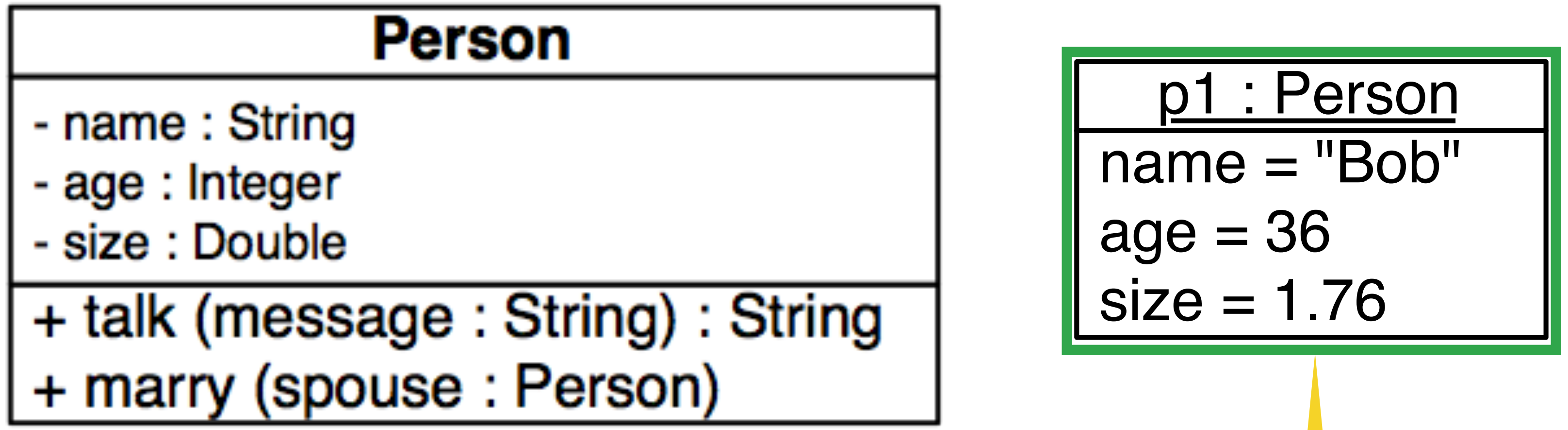
Person

- name : String
- age : Integer
- size : Double

+ talk (message : String) : String
+ marry (spouse : Person)

The class Person

OOP Terminology in Swift (2)



Person

- name : String
- age : Integer
- size : Double

+ talk (message : String) : String
+ marry (spouse : Person)

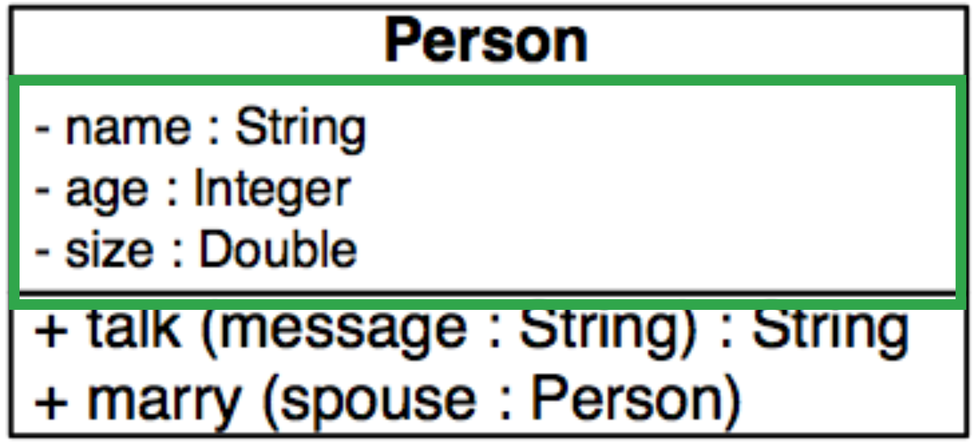
p1 : Person

name = "Bob"
age = 36
size = 1.76

An instance of the Person class

OOP Terminology in Swift (3)

- The attributes of a class are called properties



Person

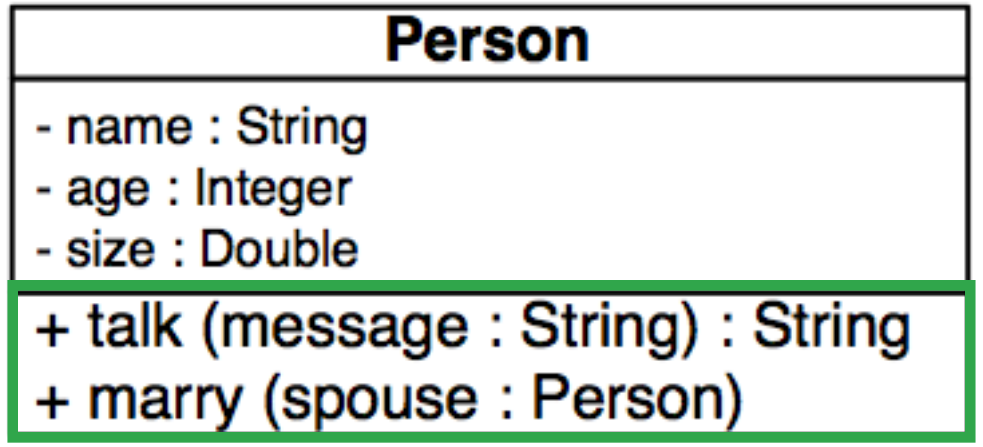
- name : String
- age : Integer
- size : Double

+ talk (message : String) : String
+ marry (spouse : Person)

Has three properties: name, age and size

OOP Terminology in Swift (4)

- The operations of a class are called methods



Person

- name : String
- age : Integer
- size : Double

+ talk (message : String) : String
+ marry (spouse : Person)

Person has two methods:
talk and marry

OOP Terminology in Swift (5)

- Classes **may** declare properties which consist of
  - A method that returns the value of the property (getters)
  - A method that set the value of the property (setters)
- To initialize the properties of a class we use the `init()` method
- Swift supports single inheritance
  - A class can inherit from one superclass
  - A class can implement many protocols
- A protocol in Swift is similar to an interface in Java

Swift vs. Java: Class definition

Temperature.swift

```swift
import Foundation

class Temperature {

    private var celsiusValue: Double

    init() {
        celsiusValue = Temperature.randomTemperature()
        print("°C: \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  (Double)(arc4random()) / 10000 % 50
    }
}
```

Temperature.java

```java
import java.util.Random;

class Temperature {

  private double celsiusValue;

  Temperature() {
        celsiusValue = randomTemperature();
        System.out.println("°C: " + celsiusValue);
  }

  static double randomTemperature(){
        Random random = new Random();
        return random.nextDouble() * 50.0;
  }

  double getCelsiusValue() {
        return celsiusValue;
  }

  void setCelsiusValue(double celsiusValue) {
        this.celsiusValue = celsiusValue;
  }
}
```

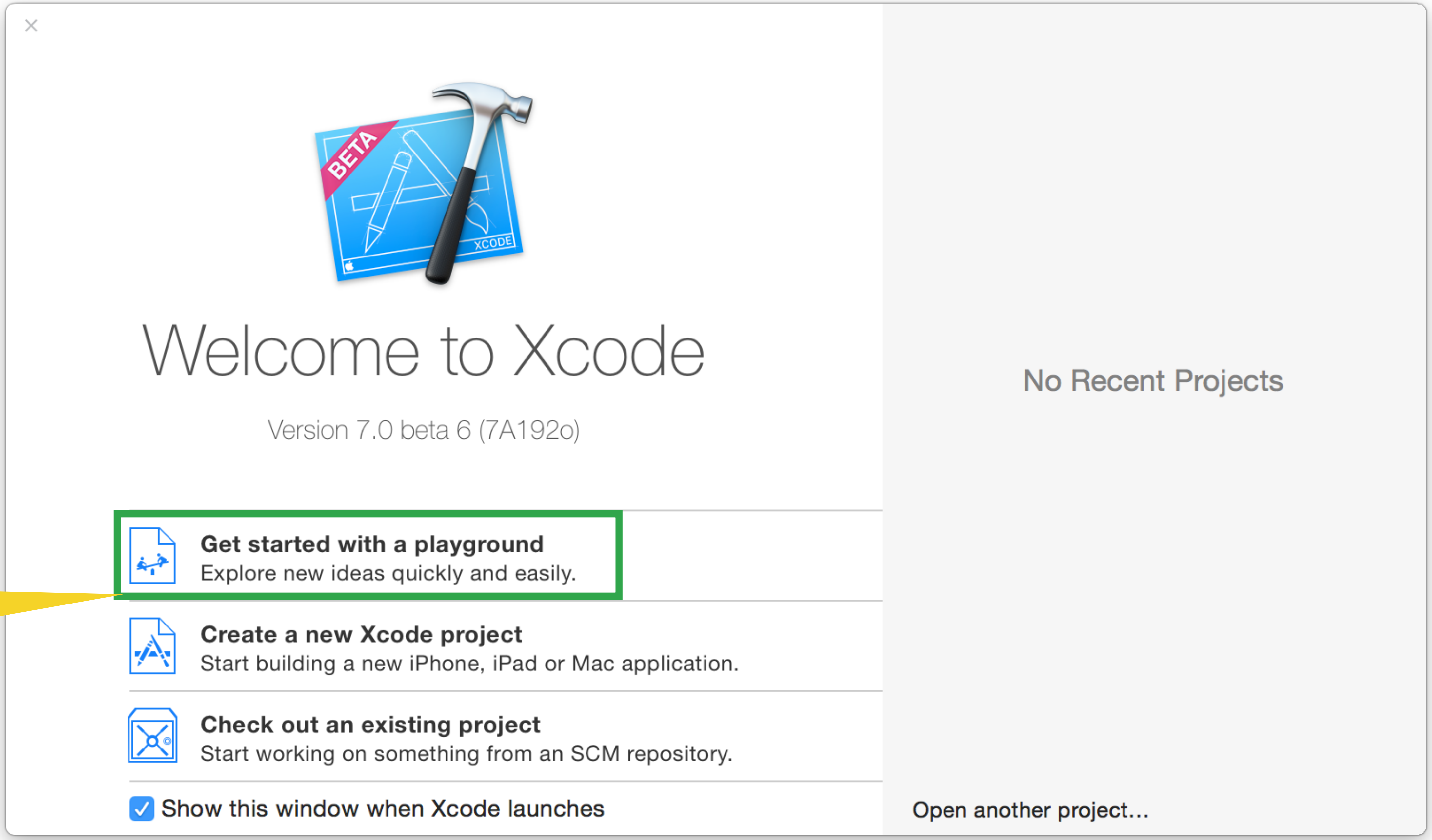
Enough talked - Lets write some code!

Xcode 7

Playgrounds

- Constantly evaluates your code as you type it
  - Whenever you finish a statement
  - Or even when you take a break :)
- Core Features
  - The sidebar immediately shows the results of the code you write
  - Quick look allows you to inspect the value of all kind of objects
- Playgrounds can be part of your daily development routine!

Xcode 7 - Create a playground



Welcome to Xcode

Version 7.0 beta 6 (7A192o)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Start building a new iPhone, iPad or Mac application.

Check out an existing project
Start working on something from an SCM repository.

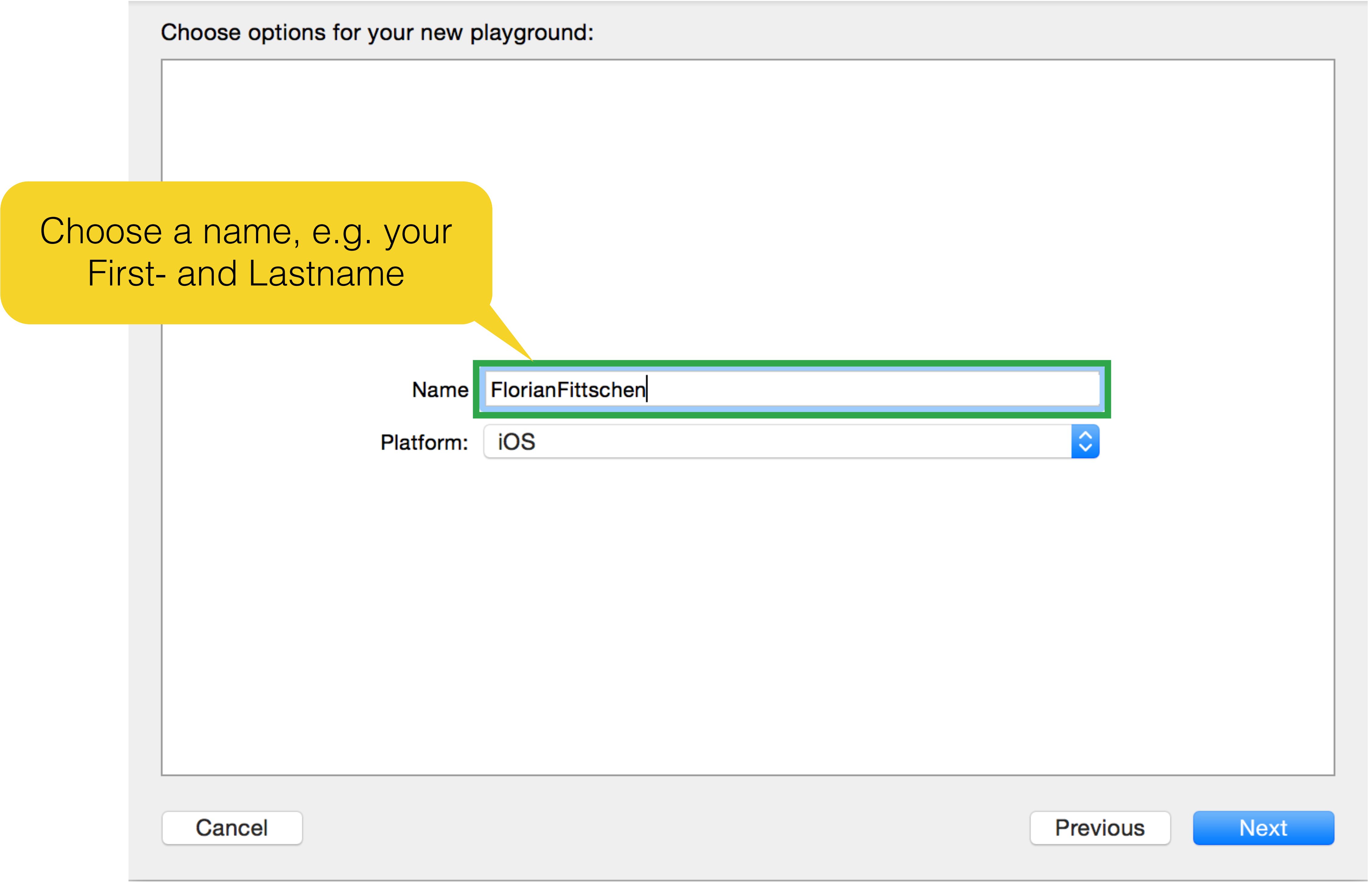
☑ Show this window when Xcode launches

No Recent Projects

Open another project...

Create a new playground

Xcode 7 - Create a playground



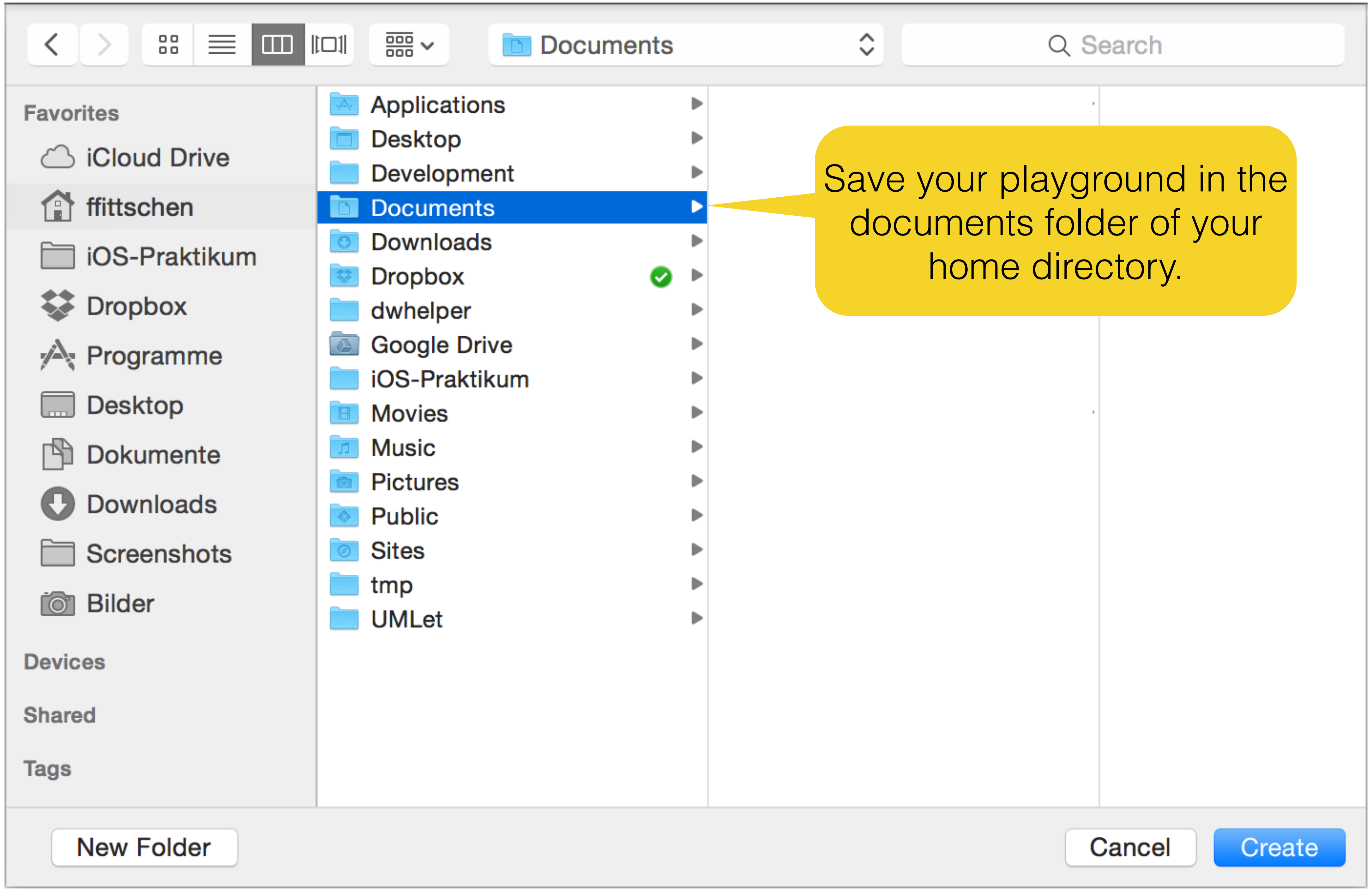
Choose options for your new playground:

Name FlorianFittschen

Platform: iOS

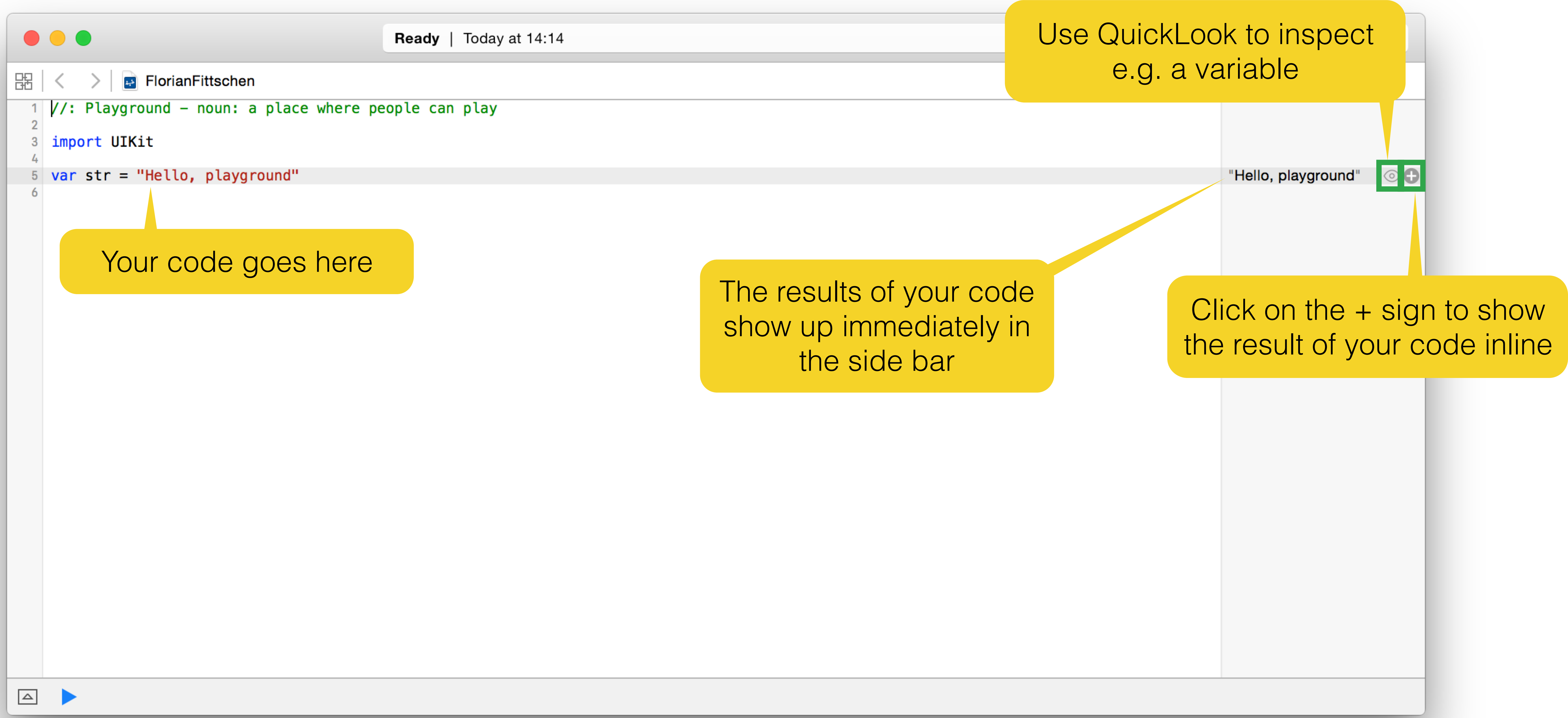
Choose a name, e.g. your First- and Lastname

Xcode 7 - Create a playground



Save your playground in the documents folder of your home directory.

Xcode 7 - Create a playground



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

import UIKit

var str = "Hello, playground"
```

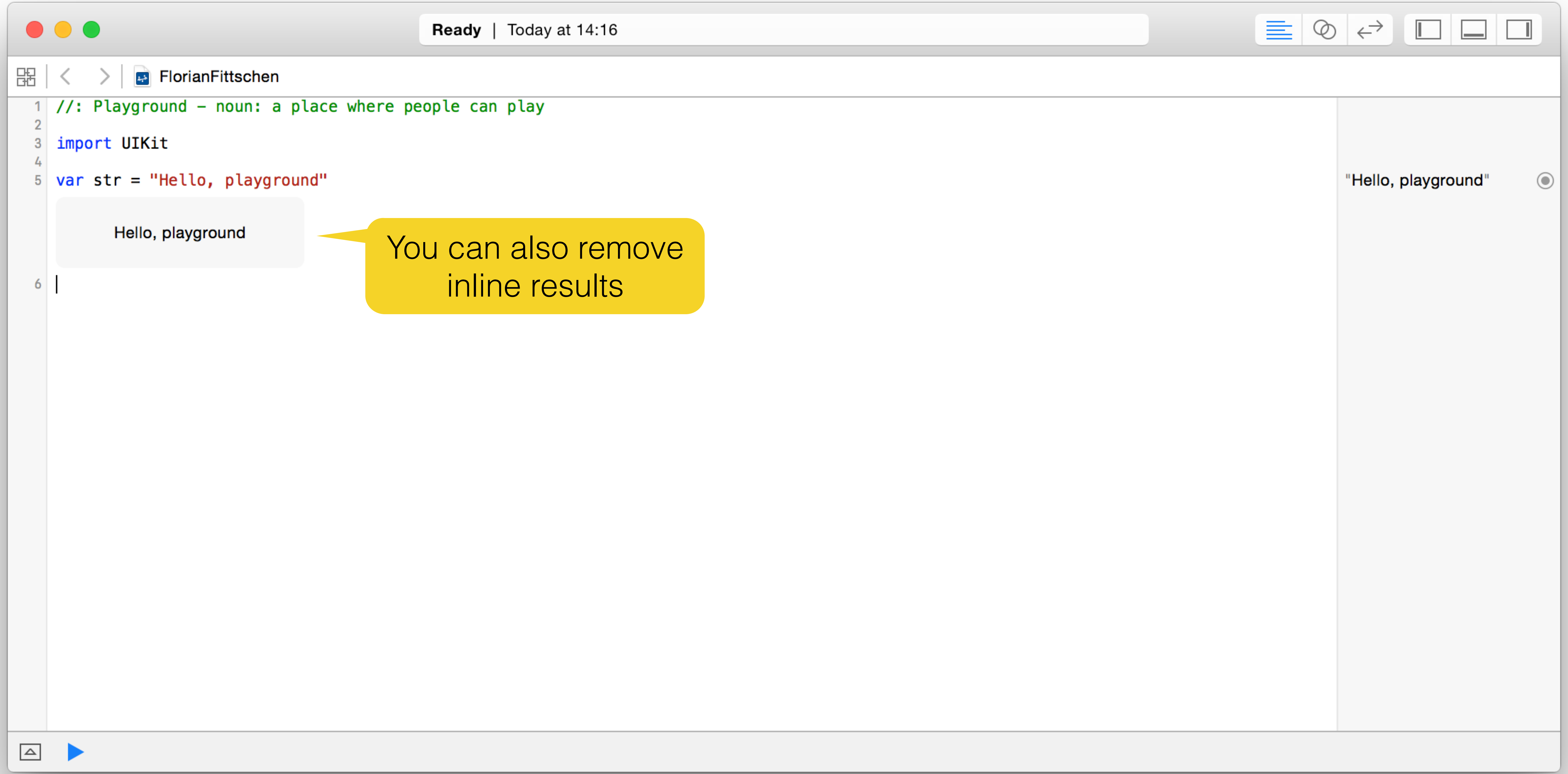
Use QuickLook to inspect e.g. a variable

Your code goes here

The results of your code show up immediately in the side bar

Click on the + sign to show the result of your code inline

Xcode 7 - Create a playground



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

import UIKit

var str = "Hello, playground"
```

You can also remove inline results

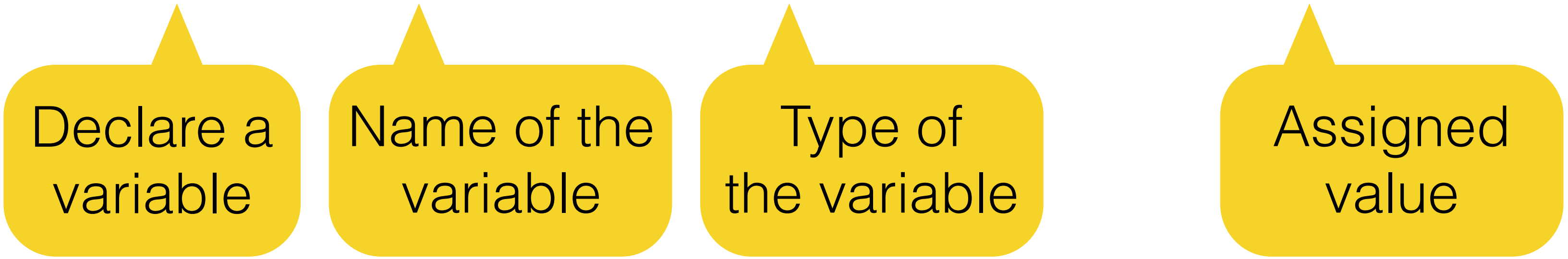
Outline

- History and Introduction
- Swift - The Basics
  - **Variables and Constants**
  - Fundamental Types
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Variables and Constants

- Variables

```swift
var personA: String = "Max Mustermann"
```



Declare a variable

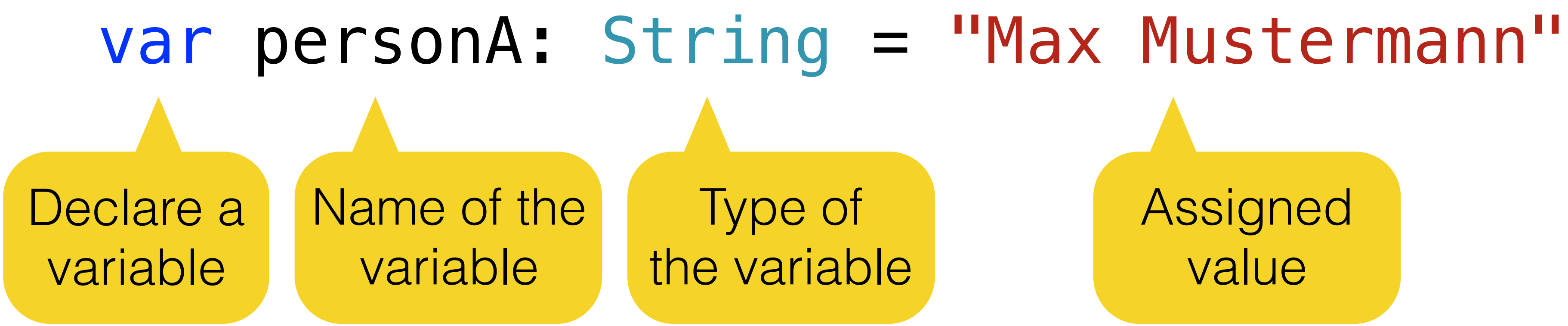
Name of the variable

Type of the variable

Assigned value

Variables and Constants

- Variables



```
var personA: String = "Max Mustermann"
```

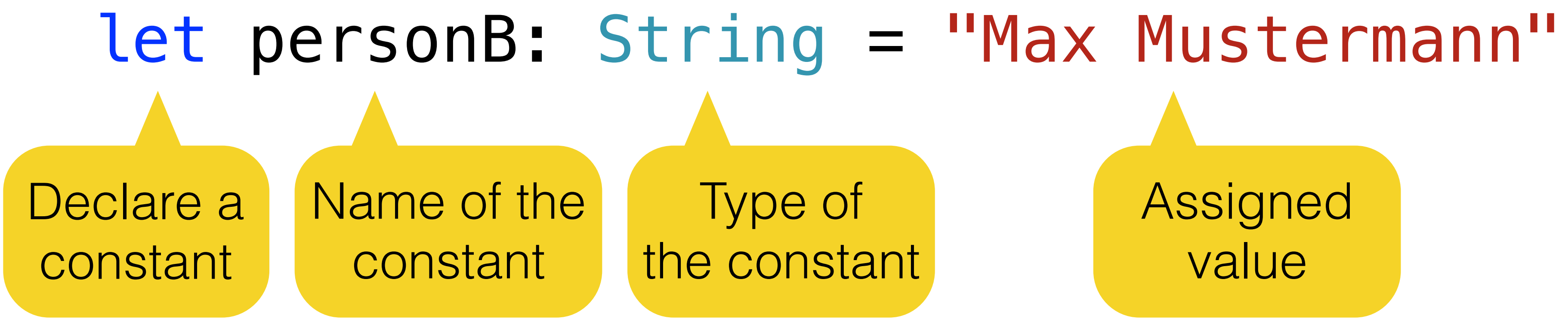
Declare a variable

Name of the variable

Type of the variable

Assigned value

- Constants



```
let personB: String = "Max Mustermann"
```

Declare a constant

Name of the constant

Type of the constant

Assigned value

Variables and Constants

- Variables

```
var personA: String = "Max Mustermann"
personA = "Peter Mustermann"
```

- Constants

```
let personB: String = "Max Mustermann"
personB = "Peter Mustermann"
```

What happens?

Variables and Constants

- Variables

```swift
var personA: String = "Max Mustermann"
personA = "Peter Mustermann"
```

- Constants

```swift
let personB: String = "Max Mustermann"
personB = "Peter Mustermann"
```

You can only set the value of a constant once

! In Swift we use constants whenever possible!

Outline

- History and Introduction
- Swift - The Basics
  - Variables and Constants
  - **Fundamental Types**
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Fundamental Types

```
// Int
let age: Int = 23

// Double
let expectedGrade: Double = 1.0

// Bool
let motivated: Bool = true

// String
let person: String = "Max Mustermann"
```

Task: Define your own variables (`var`) and constants (`let`)

Type Inference

- In Swift we do not need to specify a type most of the time
- Example: `let person = "Max Mustermann"`
- Because we assign a `String` value, Swift „infers“ the type of person to be `String`

Important:

- Swift is a explicitly typed language
- Type checks are done at compile time
- Type inference is for your convenience and keeps code readable while preserving all the benefits of explicit typing

Fundamental Types (Inferred)

```
// Int
let age = 23

// Double
let expectedGrade = 1.0

// Bool
let motivated = true

// String
let person = "Max Mustermann"
```

Outline

- History and Introduction
- Swift - The Basics
  - Variables and Constants
  - Fundamental Types
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Strings

- A `String` is a buffer of Unicode characters
- As we learned, a `String` can be created quickly using literals, e.g.

```
let person = "Max Mustermann"
```

- **Task:** Use the count method on the characters property of a String

```
person.characters.count
```

- **Task:** Use the **+** operator to concatenate Strings

```
let firstName = "Max"                 // "Max"
let lastName = "Mustermann"           // "Mustermann"
let name = firstName + " " + lastName // "Max Mustermann"
```

String Interpolation

- You can create a new String from a mix of constants, variables or literals

```
let person = "Max Mustermann"
let age = 23
```

- Lets create a new String out of these values

```
let introduction = "My name is " + person + ". I'm \(age) years old"
```

String Interpolation by putting e.g. an Int into \()

- The constant `introduction` evaluates to:

```
"My name is Max Mustermann. I'm 23 years old"
```

Xcode 7 - Open the playground for exercises

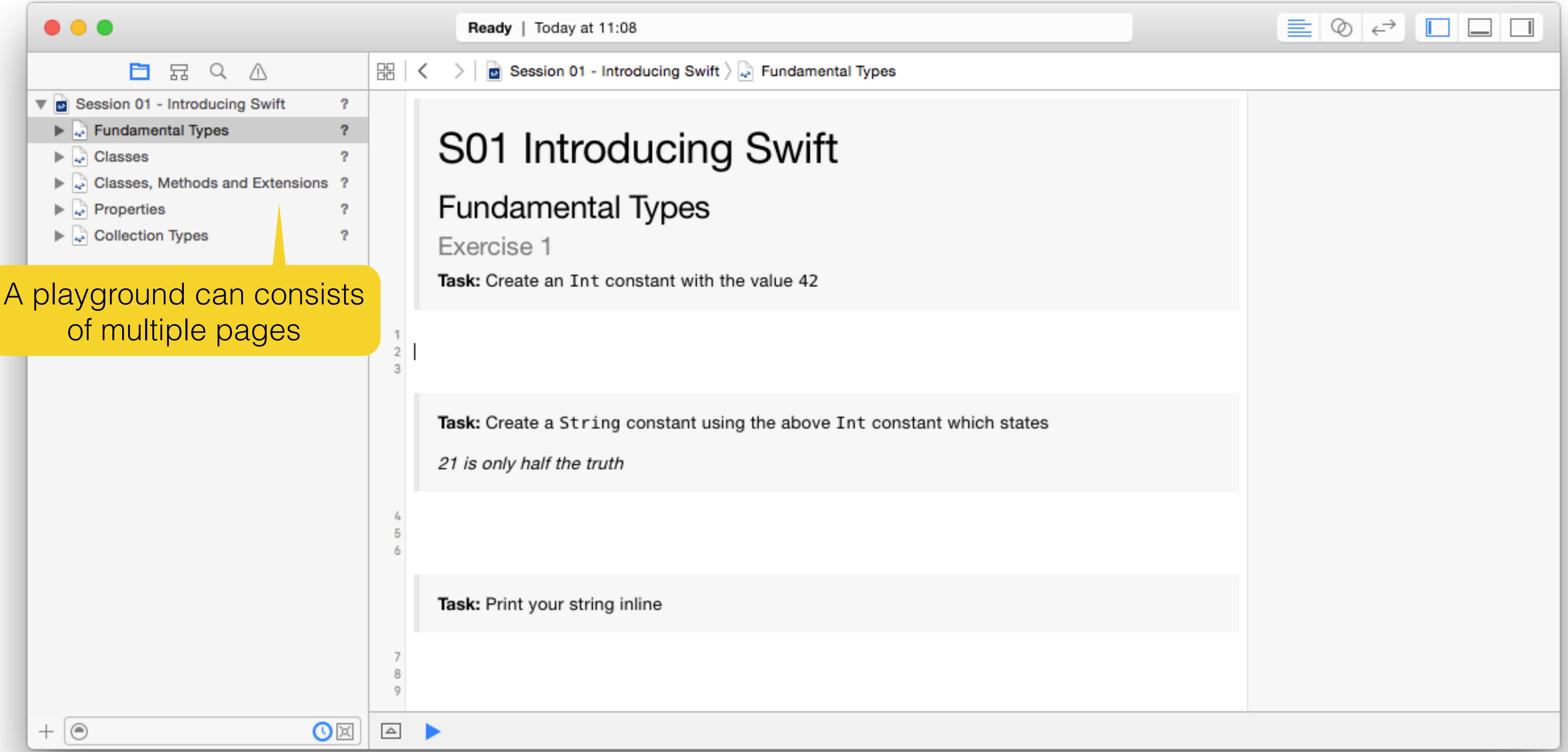
- You already have your own Playground to play with Swift Code
- For the exercises we use playgrounds we prepared for this session.

They already include code snippets and you do not have to type everything on your own.
You ask why?
Because we like you.

Task

Open the “**Session 01.playground**“
in the **exercise folder** of Session 01.

Xcode 7 - Playground for exercises



Ready | Today at 11:08
Session 01 - Introducing Swift 〉 Fundamental Types
Session 01 - Introducing Swift ?
Fundamental Types ?
Classes ?
Classes, Methods and Extensions ?
Properties ?
Collection Types ?

A playground can consists of multiple pages

S01 Introducing Swift

Fundamental Types

Exercise 1

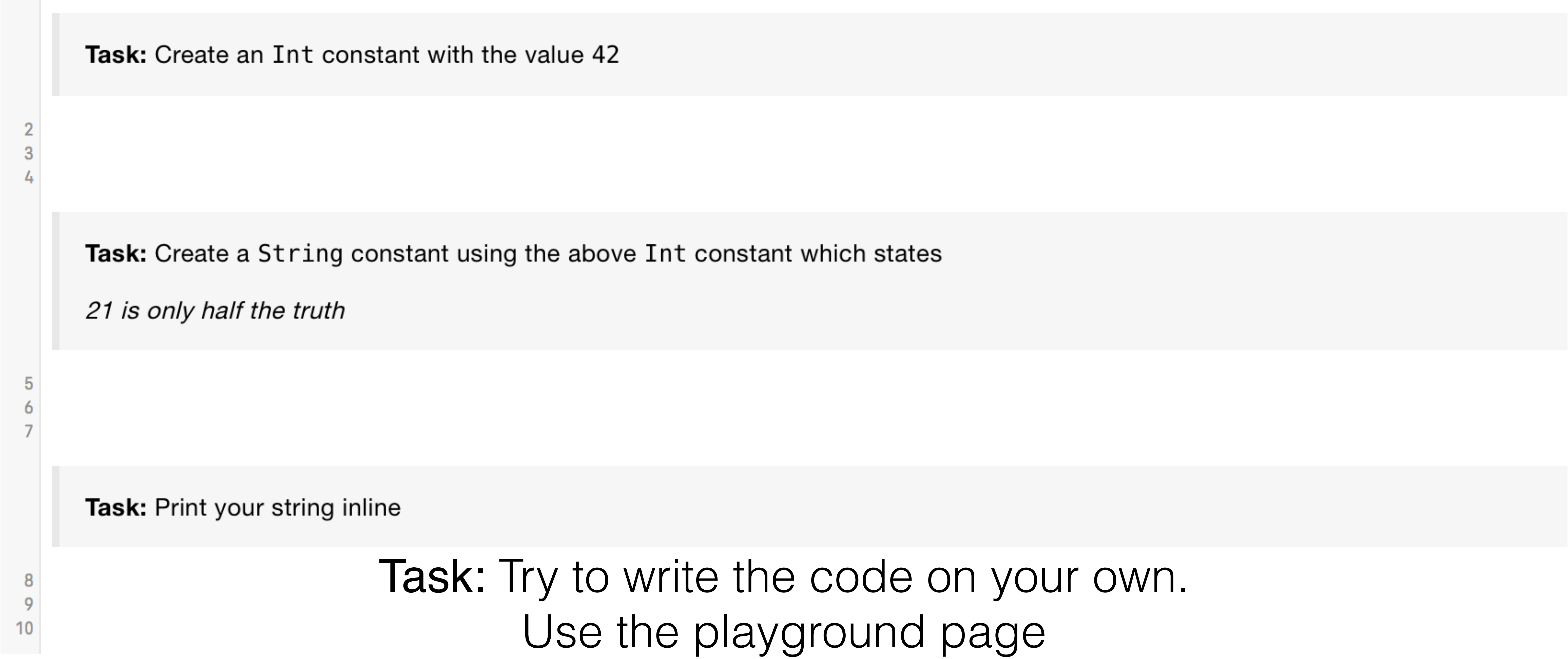
Task: Create an `Int` constant with the value 42

Task: Create a `String` constant using the above `Int` constant which states

21 is only half the truth

Task: Print your string inline

Strings - **Exercise 1**



Task: Create an `Int` constant with the value 42

Task: Create a `String` constant using the above `Int` constant which states

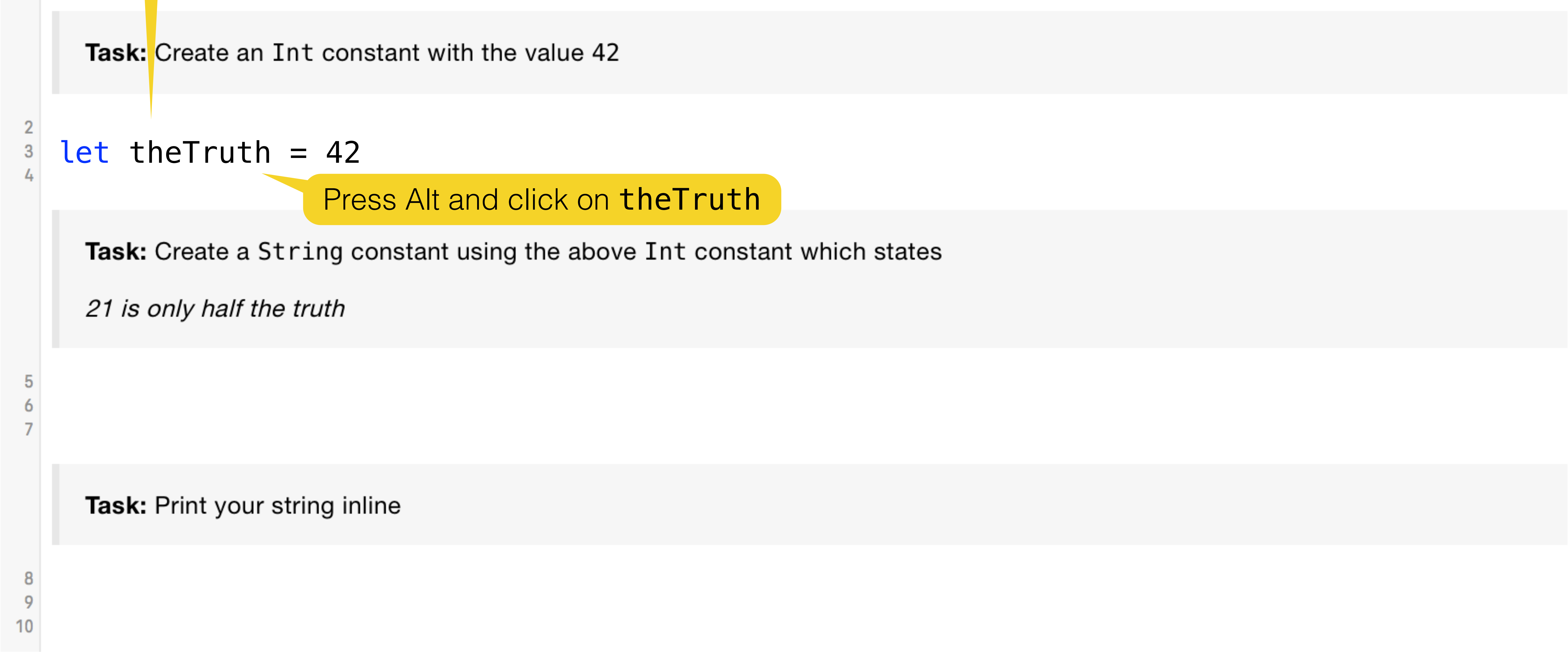
21 is only half the truth

Task: Print your string inline

Task: Try to write the code on your own.
Use the playground page
“Fundamental Types“

Strings - **Exercise 1**

Type Int is inferred



Task: Create an `Int` constant with the value 42

```swift
let theTruth = 42
```

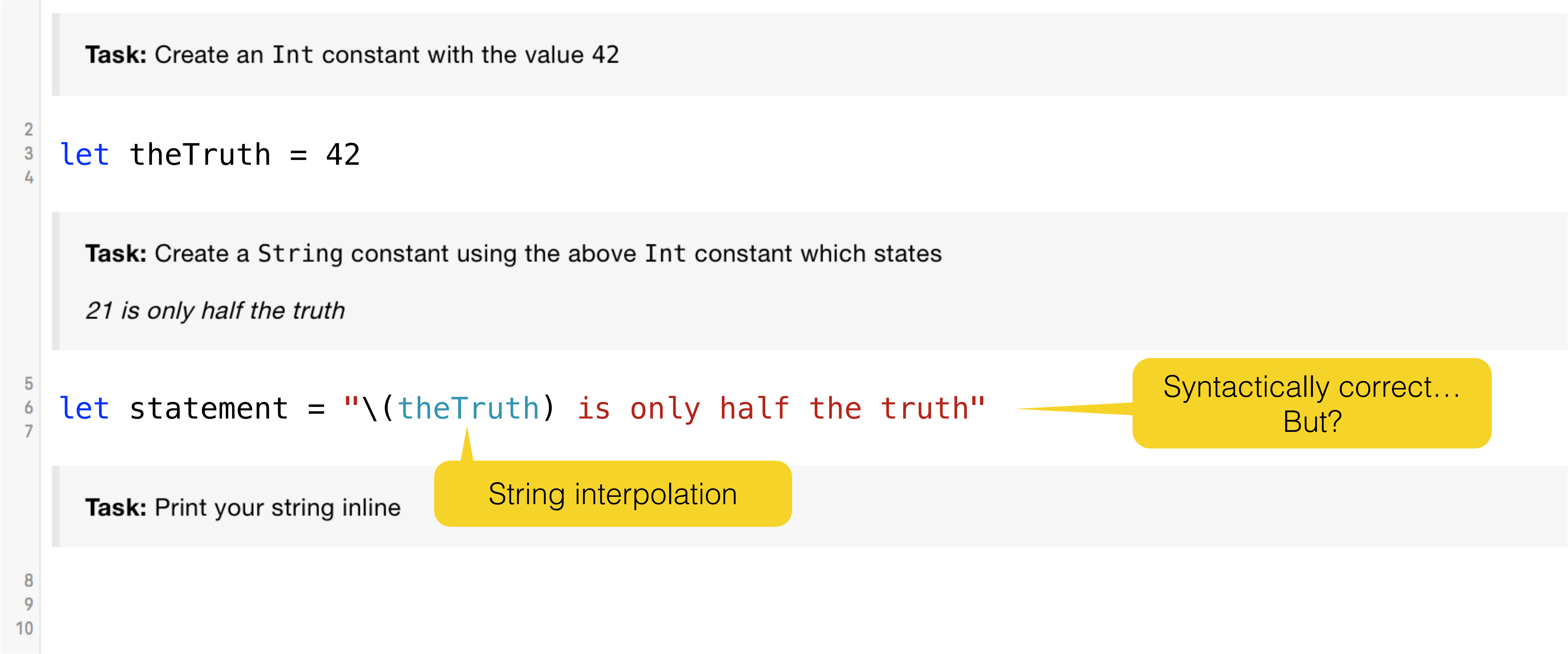
Press Alt and click on `theTruth`

Task: Create a `String` constant using the above `Int` constant which states

21 is only half the truth

Task: Print your string inline

Strings - **Exercise 1**



Task: Create an `Int` constant with the value 42

```swift
let theTruth = 42
```

Task: Create a `String` constant using the above `Int` constant which states

21 is only half the truth

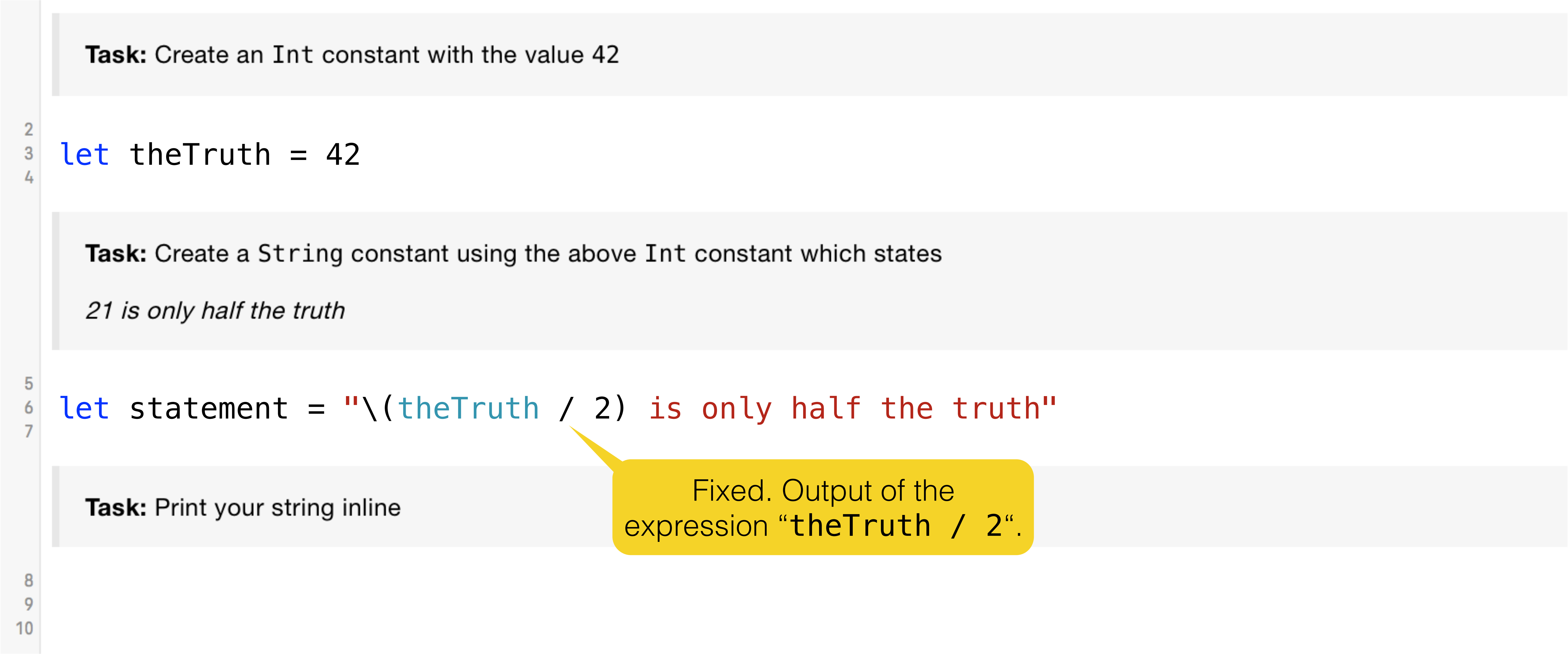
```swift
let statement = "\(theTruth) is only half the truth"
```

Task: Print your string inline

Syntactically correct…
But?

String interpolation

Strings - **Exercise 1**



Task: Create an `Int` constant with the value 42

```
let theTruth = 42
```

Task: Create a `String` constant using the above `Int` constant which states

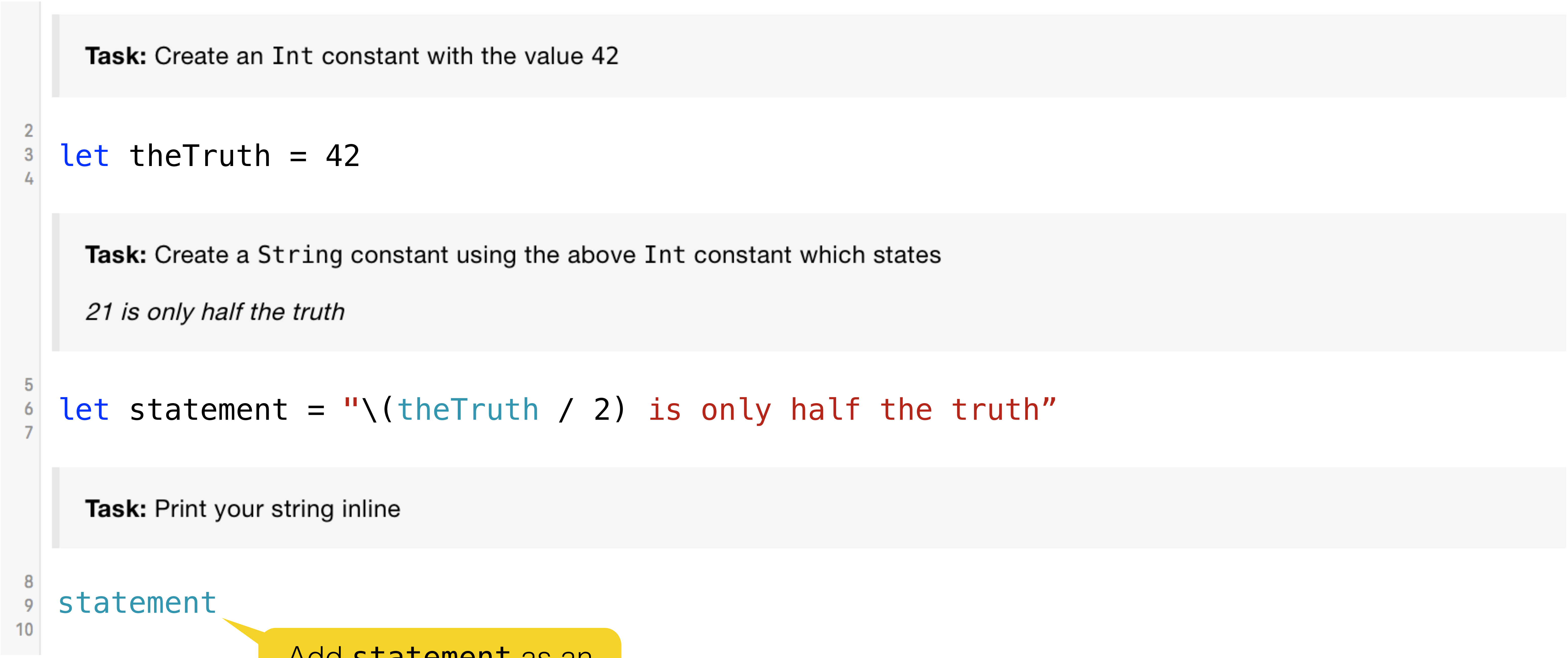
21 is only half the truth

```
let statement = "\(theTruth / 2) is only half the truth"
```

Task: Print your string inline

Fixed. Output of the expression “`theTruth / 2`“.

Strings - **Exercise 1**



Task: Create an `Int` constant with the value 42

```
let theTruth = 42
```

Task: Create a `String` constant using the above `Int` constant which states

21 is only half the truth

```
let statement = "\(theTruth / 2) is only half the truth”
```

Task: Print your string inline

```
statement
```

Add **`statement`** as an inline result

Outline

- History and Introduction
- Swift - The Basics
  - Variables and Constants
  - Fundamental Types
  - Strings
  - **Classes (Initializers, Methods, Properties, Access Control)**
  - Collection Types
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Classes

To declare a Swift class you need to

- Choose an appropriate **class name**
- Define **inheritance** from a superclass
- State the **protocols** your class will implement
- Define private and public **properties**
- Define private and public **methods**

Classes

Class name

```
class ClassName: SuperClass, SomeProtocol {

    // properties

    // initializers

    // methods

}
```

Keyword for declaring a new class

Superclass to inherit from

Protocol to conform to

A class declaration always starts and end with curly brackets

Classes - Exercise 2

Class name

```
class Temperature {




}
```

Classes - **Exercise 2**

Class name

```
class Temperature {

    private var celsiusValue = 0.0







}
```

Definition and initialization of a property

Type **`Double`** is inferred as we initialize the property with **`0.0`**

Classes - **Exercise 2**

Class name

Definition and initialization of a property

Definition of the default initializer

```
class Temperature {

    private var celsiusValue = 0.0

    init() {


    }



}
```

Classes - **Exercise 2**

Class name

Definition and initialization of a property

Definition of the default initializer

Wait. There is no randomTemperature() method yet

```
class Temperature {

    private var celsiusValue = 0.0

    init() {
        celsiusValue = randomTemperature()
        print("°C \(celsiusValue)")
    }



}
```

Use of string interpolation to print the celsiusValue

Classes - **Exercise 2**

Class name

Definition and initialization of a property

Definition of the default initializer

Definition of an class („static“) method

Does not work. Why?

```
class Temperature {

    private var celsiusValue = 0.0

    init() {
        celsiusValue = randomTemperature()
        print("°C \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  42.0
    }
}
```

Classes - **Exercise 2**

Class name

Definition and initialization of a property

Definition of the default initializer

Definition of an class („static“) method

```
class Temperature {

    private var celsiusValue = 0.0

    init() {
        celsiusValue = Temperature.randomTemperature()
        print("°C \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  42.0
    }
}
```

Thats not really random :)

Classes - **Exercise 2**

Class name

Definition and initialization of a property

Definition of the default initializer

Definition of an class („static“) method

```
class Temperature {

    private var celsiusValue = 0.0

    init() {
        celsiusValue = Temperature.randomTemperature()
        print("°C \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  (Double)(arc4random()) / 10000 % 50
    }
}
```

Casting: More about that in Session 06.

Creates a random temperature

Classes - **Exercise 2**

Class name

No need to initialize because we do this in the init() method. We just define that celsiusValue is of type Double.

Definition and initialization of a property

Definition of the default initializer

Definition of an class („static“) method

```
class Temperature {

    private var celsiusValue: Double

    init() {
        celsiusValue = Temperature.randomTemperature()
        print("°C \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  (Double)(arc4random()) / 10000 % 50
    }
}
```

Classes - **Exercise 2**

```
class Temperature {
    //...
}

let temperatureInstance = Temperature()

temperatureInstance.celsiusValue
```

Create a new instance of our Temperature class.

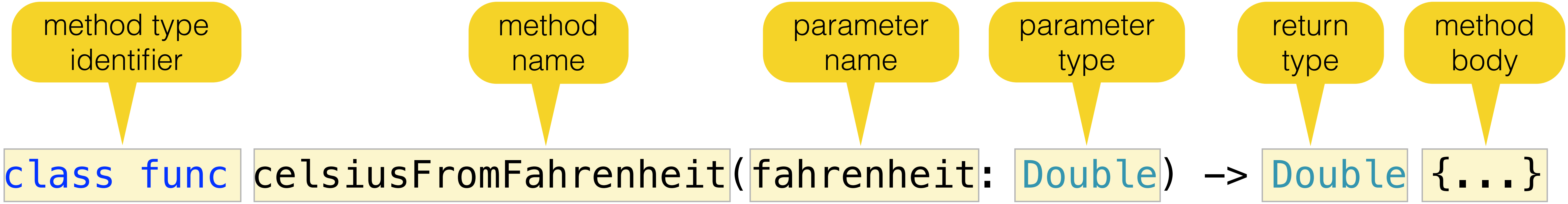
Show the celsiusValue in the sidebar

Functions and Methods

- A function can be declared inside or outside a class using the keyword `func`
- If a function is declared inside a class we call it method
- Both can consist of:
  - 0..n parameters (separated by colons)
  - 0..n return types (more about that in Session 03)
- Class („static“) methods are defined using the keyword `class func`

Methods

Let’s inspect the `celsiusFromFahrenheit` method in more detail



method type identifier

method name

parameter name

parameter type

return type

method body

```
class func celsiusFromFahrenheit(fahrenheit: Double) -> Double {...}
```

Functions and Methods

```
class Temperature {
    class func calcTempMean(tempOne: Double, tempTwo: Double ) -> Double {
        return (tempOne + tempTwo) / 2
    }
}

func meanTemperature(tempOne: Double, tempTwo: Double ) -> Double {
    return (tempOne + tempTwo) / 2
}

let myFirstMeanTemperature = Temperature.calcTempMean(10, tempTwo: 20)

let mySecondMeanTemperature = meanTemperature(10, tempTwo: 20)
```

Method

Function

You need to name the parameters starting with the 2nd

! In Swift 2 methods and functions behave pretty much the same.

Methods - **Exercise 3**

We now add two more methods to the temperature class

```
(1) func fahrenheitValue() -> Double {...}

(2) class func celsiusFromFahrenheit(fahrenheit: Double) -> Double {...}
```

Methods - **Exercise 3**

```
extension Temperature {

    func fahrenheitValue() -> Double {
        let fahrenheitValue = ((celsiusValue * 9.0) / 5.0) + 32.0
        return fahrenheitValue
    }






}
```

Instance Method
parameter: None,
return type: Double

Methods - **Exercise 3**

```
extension Temperature {

    func fahrenheitValue() -> Double {
        let fahrenheitValue =  ((celsiusValue * 9.0) / 5.0) + 32.0
        return fahrenheitValue
    }

    class func celsiusFromFahrenheit(fahrenheit: Double) -> Double {
        let celsiusTemp = (fahrenheit - 32.0) * (5.0 / 9.0)
        return celsiusTemp
    }
}
```

Instance Method
parameter: None,
return type: Double

Class Method
parameter: Double,
return type: Double

Task:

Create an instance of Temperature and try to invoke fahrenheitValue()

Extensions

- An extension allows you to extend the functionality of a class
- The new functionality defined in an extension will be available on all existing instances of that class in this module

Extensions allow you to

- Split up a large class into several files
- Extend functionality of existing types without the need for building a subclass

We talk in more detail about extensions in Session 06 - Advanced Swift.

Properties

- Syntactical shorthand for declaring (and implementing) accessor methods
- A class may declare properties which consist of
  - A method that returns the value of instance variables (getters)
  - A method that sets the value of an instance variable (setters)
- There are two types of properties in Swift
  - Stored properties: A constant or variable that is stored as part of an instance of a class. This actually requires additional memory
  - Computed properties: Do not actually store a value, instead provide a getter to retrieve and optionally a setter to modify other properties or values indirectly

Attributes in Java

- In order to enforce information hiding, you typically define attributes with the lowest access level
- However accessor methods allow to retrieve and modify their values from the class

Temperature.java

```
import java.util.Random;

class Temperature {
   private double celsiusValue;

   Temperature () {
        celsiusValue = randomTemperature();
        System.out.println("celsius: " + celsiusValue);
   }

   public double getCelsiusValue() {
        return celsiusValue;
   }

   public void setCelsiusValue(double celsiusValue)
        this.celsiusValue = celsiusValue;
   }
}
```

Getter

Setter

Stored Properties - **Example**

```
class Temperature {

    private var celsiusValue: Double

    init() {
        celsiusValue = Temperature.randomTemperature()
        print("°C: \(celsiusValue)")
    }

    class func randomTemperature() -> Double {
        return  (Double)(arc4random()) / 10000 % 50
    }
}
```

Stored property. Default getter and setter created by the compiler.

Computed Properties - **Exercise 4**

```
extension Temperature {

    var description: String {
        get {
            return "°C \(celsiusValue)"
        }
        set {

        }

    }
}
```

Lets add a computed property called `description`.

Computed Properties - **Exercise 4**

```
extension Temperature {

    var description: String {
        get {
            return "°C \(celsiusValue)"
        }
    }
}
```

Lets add a computed property called `description`.

We can leave out the setter when we do not need one

Computed Properties - **Exercise 4**

```
extension Temperature {

    var description: String {
        return "°C \(celsiusValue)"
    }
}
```

Even shorter version of a computed property

Swift vs. Java: Access Control

Swift

- Scope-based
  Encourages organization of your code (into different .swift files)
- Keywords
  - `public` Everywhere
  - `internal` Same module (*Default*)
  - `private` Same file

Java

- Type-based
  Encourages the use of types
- Keywords
  - `public` Everywhere
  - `protected` Same Class, same package, same hierarchy
  - *No modifier* Same Class, same package (*Default*)
  - `private` Same class

Access Control - **Example**

1

```swift
private class Temperature {…}
```

Class is private, only accessible from functions and classes defined in the same file

2

```swift
internal func getName() {…}
```

3

```swift
private class SomePrivateClass {

}

func someFunction() -> SomePrivateClass {
    let myClass = SomePrivateClass()
    return myClass
}
```

Access Control - **Example**

1

```
private class Temperature {…}
```

Class is private, only accessible from functions and classes defined in the same file

2

```
internal func getName() {…}
```

Function is internal, only accessible from classes included in the same module

3

```
private class SomePrivateClass {

}

func someFunction() -> SomePrivateClass {
    let myClass = SomePrivateClass()
    return myClass
}
```

Access Control - **Example**

1

```
private class Temperature {…}
```

Class is private, only accessible from functions and classes defined in the same file

2

```
internal func getName() {…}
```

Function is internal, only accessible from classes included in the same module

3

```
private class SomePrivateClass {

}

func someFunction() -> SomePrivateClass {
    let myClass = SomePrivateClass()
    return myClass
}
```

Returns an error. Why?

Access Control - **Example**

1

```swift
private class Temperature {…}
```

Class is private, only accessible from functions and classes defined in the same file

2

```swift
internal func getName() {…}
```

Function is internal, only accessible from classes included in the same module

3

```swift
private class SomePrivateClass {

}

private func someFunc() -> SomePrivateClass {
    let myClass = SomePrivateClass()
    return myClass
}
```

someFunc is internal by default but returns a private type.

Public getter and private setter - **Example**

```swift
— Temperature.swift —

class Temperature {

    private(set) var celsiusValue: Int = 0

}
```

celsiusValue can only be set inside the file where the Temperature class was declared.

Public getter and private setter - **Example**

```
— Temperature.swift —

class Temperature {

    private(set) var celsiusValue: Int = 0

}
```

celsiusValue can only be set inside the file where the Temperature class was declared.

```
— AppDelegate.swift —

let myTemp = Temperature()

let myCelsiusValue = myTemp.celsiusValue
myTemp.celsiusValue = 2
```

Throws an error because setter of celsiusValue is declared private.

Outline

- History and Introduction
- Swift - The Basics
  - Variables and Constants
  - Fundamental Types
  - Strings
  - Classes (Initializers, Methods, Properties, Access Control)
  - **Collection Types**
- Get used to Xcode 7 Playgrounds (on the go)
- Exercise

Collection Types - Array

- An array is a container, which contains **0 to n** elements
- Example

```
var partyParticipants = ["Stephan", "Lucas", "Jan", "Barbara", "Martin"]

partyParticipants[1] = "Lukas"
```

“Subscript“ Notation: Access the second element of the array and replace it.

Comma separated list of Strings. Array type is inferred: `[String]`

- How to create an array with 6 `Int` values

```
var lostNumbers = [4,8,15,16,23,42]
```

Collection Types - Array

Common methods/properties

- `var count: Int { get }`
  Returns the number of elements in the array
- `func append(newElement: Element)`
  Adds **`newElement`** of type **`Element`** at the end of the array
- `func insert(newElement: Element, atIndex i: Int)`
  Inserts **`newElement`** at index **`i`**. If **`i`** is occupied, elements from **`i`** will be shifted.
- `func removeAtIndex(index: Int) -> Element`
  Removes the object at **`index`** from the array

First sign of generics in Swift. More in Session 06.

Collection Types - Array - **Exercise 5**

Which type has party participants?

```
// Participants of the tea party
var partyParticipants = ["Stephan", "Lucas", "Jan",
"Barbara", "Martin"]

// Correct the spelling of Lucas
partyParticipants[1] = "Lukas"

// Add yourself to the guest list


// Show the array of participants in the sidebar
```

Task: Try the next two steps on your own. Use the Collection Types page in your Session 01 playground.

Collection Types - Array - **Exercise 5**

```swift
// Participants of the tea party
var partyParticipants = ["Stephan", "Lucas", "Jan",
"Barbara", "Martin"]

// Correct the spelling of Lucas
partyParticipants[1] = "Lukas"

// Add yourself to the guest list
partyParticipants.append("Florian")

// Show the array of participants in the sidebar
```

Add an element at the end of the array

Collection Types - Array - **Exercise 5**

```
// Participants of the tea party
var partyParticipants = ["Stephan", "Lucas", "Jan",
"Barbara", "Martin"]

// Correct the spelling of Lucas
partyParticipants[1] = "Lukas"

// Add yourself to the guest list
partyParticipants.append("Florian")

// Show the array of participants in the sidebar
partyParticipants
```

Show the content of `partyParticipants` in the sidebar

Collection Types - Dictionary

- An dictionary is a container, which contains **0 to n** key/value pairs
  - **Key** can be of any type that conforms to the **`Hashable`** protocol
  - **Value** can be any type
- Example

```
var teaOrders = [
    "Stephan": "Earl Grey",
    "Lukas"  : "Green Tea",
    "Jan"    : "Black Tea",
    "Barbara": "Earl Grey“,
    "Martin" : "Green Tea"
]

teaOrders["Barbara"] = "Earl Grey with a drop of milk"
```

which all fundamental types do already

5 Key/value pairs. Key and value are Strings.

Our dictionary checks if the key already exists. If yes the value is replaced, if not a new key/value pair is created.

Collection Types - Dictionaries

Common methods/properties

- `var count: Int { get }`
  Returns the number of key-value pairs in the dictionary

First sign of optionals, see Session 03

- `func updateValue(value: Value, forKey key: Key) -> Value?`
  Inserts or updates a `value` for a given `key` and returns the previous value for that key.

- `func removeValueForKey(key: Key) -> Value?`
  Removes the key-value pair for the specified `key` from the dictionary and returns the previous value for that key.

Collection Types - Dictionaries - **Exercise 6**

```
// Dictionary of participants and their tea orders
var teaOrders = [
   "Stephan": "Earl Grey",
   "Lukas"  : "Green Tea",
   "Jan"    : "Black Tea",
   "Barbara": "Earl Grey",
   "Martin" : "Green Tea"
]

// Barbara changed her mind.
teaOrders["Barbara"] = "Earl Grey with a drop of milk"

// Place your order


// Show the updated dictionary of tea orders in the sidebar
```

Collection Types - Dictionaries - **Exercise 6**

```
// Dictionary of participants and their tea orders
var teaOrders = [
   "Stephan": "Earl Grey",
   "Lukas"  : "Green Tea",
   "Jan"    : "Black Tea",
   "Barbara": "Earl Grey",
   "Martin" : "Green Tea"
]

// Barbara changed her mind.
teaOrders["Barbara"] = "Earl Grey with a drop of milk"

// Place your order
teaOrders.updateValue("Black Tea", forKey: "Johannes")

// Show the updated dictionary of tea orders in
```

Long version as an alternative to the subscript notation

Collection Types - Dictionaries - **Exercise 6**

```swift
// Dictionary of participants and their tea orders
var teaOrders = [
   "Stephan": "Earl Grey",
   "Lukas"  : "Green Tea",
   "Jan"    : "Black Tea",
   "Barbara": "Earl Grey",
   "Martin" : "Green Tea"
]

// Barbara changed her mind.
teaOrders["Barbara"] = "Earl Grey with a drop of milk"

// Place your order
teaOrders.updateValue("Black Tea", forKey: "Johannes")

// Show the updated dictionary of tea orders in the sidebar
teaOrders
```

Show the content of `teaOrders` in the sidebar

Summary

You learned…

- the fundamental data types
- how to deal with Strings
- how to declare and define classes
- how to declare and define methods
- how to invoke methods
- how to create objects
- how to initialize objects
- how to use arrays and dictionaries
- how to use Xcode Playgrounds

For More Information

- WWDC 14 Video + Slidedeck: Introduction to Swift
- iBook: The Swift Programming Language

Exercise Submission

For now just finish but do not commit the following exercise.

We take care of the submission in Session 02 :)

Exercise

- Create a new playground with the name „S01 YourName“
- Create a class `Cookie` with two properties `type: String` and `brand: String`
- When initializing a new Cookie instance, we want to set both properties with parameters passed to the `init` method.
- Create a class `CookieMonster`
  - `CookieMonster` has two properties `name: String` and `cookies: [Cookie]`
  - When initializing a new CookieMonster, we want to set its name with a parameter passed to the init method.
  - `CookieMonster` has an instance method `takeCookie`
    - `takeCookie` takes a Cookie as a parameter and returns nothing
    - `takeCookie` adds a Cookie to the cookies array
  - `CookieMonster` has an instance method `eatCookies`
    - `eatCookies` has no parameters and returns nothing
    - `eatCookies` prints „$NumberOfCookies$ Cooookies!!!! Om nom nom…“
    - `eatCookies` removes all cookies from the cookies array
- Create an instance of a CookieMonster, give it the name „Monster“, feed him 3 different cookies and let him eat them all
- Optional challenge: Use a for-in loop for feeding the cookies and add an additional „nom nom“ for each cookie eaten to the existing print() statement.