Study diary iOS_JingjingYang

Exercise 1

> What role does the Swift programming language have in iPhone development?

Swift is a robust and intuitive programming language created by Apple for building apps for iOS, Mac, Apple TV, and Apple Watch. Swift has become the go-to pro- gramming language for building apps across the Apple ecosystem.

While Apple is pushing Swift, you can also program iOS in Objective-C. They are the two main languages that power iOS. You can use other languages to code iOS apps, but they may require significant workarounds that require more effort than needed.

> What other programming language(s) are you familiar with?

JavaScript, Java, C++, SQL

> Based on “The Swift Programming Language” by Apple, and other sources, can you see any obvious differences to those you already know?

Swift is a fast and efficient language that provides real-time feedback. It’s de- signed to give developers more freedom than ever to write clean-looking code. One feature is known as Type Inference. Type inference means you don’t need to explicitly specify the type of a variable or constant in most cases. That makes your code cleaner and faster.

Another great feature of Swift is its syntax. Because there are no semicolons, calls to self, or parentheses around if statements, the process of typing a lot of code can “flow” much easier.

Exercise 2

> What is Xcode, and what is it used for?

Xcode is the Integrated Development Environment (IDE) used to design, develop, and publish iOS apps. Xcode contains great tools like project manager, code ed- itor, built-in documentation, debugging tools, and the Interface Builder.

It is the only supported way to develop apps by Apple, and it’s the only way you can deploy your app code to the App Store once you’re finished.

> How can you get Xcode and use it for its intended purpose?

Xcode is available as a free download from the Mac App Store.

Xcode can run only on macOS, users will need a Mac - MacBook, iMac, or Mac mini - in order to develop the iOS app using Xcode.

> What other ways of programming in Swift are there apart from using Xcode, especially if you don’t have access to a Mac computer?

1. Swift can be accessed through a fairly full-featured online IDE http://online.swiftplayground.run

https://swiftfiddle.com/

2. Install macOS Onto a Virtual Machine on Windows PC
3. Rent a Cloud-Based Mac

Exercise 3

> What is the difference between let and var in Swift?

Keyword let is to declare a constant - immutable values. Keyword var is to declare a variable - mutable values.

> Why and when would you use one or the other?

Using constant values is meaningful when you want to declare variables that are not supposed to change. Once a constant is initialized with a value, you can't change it.

If you have to change the value after initialization, use variables.

Exercise 4

> The following Swift code defines a variable called age, with the value of 26.
> var age = 26
> What would be the appropriate Swift code to display the text “You are 26 years old”?If the value of age is changed to 42, like this:
> age = 42
> the very same code should display “You are 42 years old” without any changes.
> How can you achieve this? (Hint: Look up “string interpolation” in the Swift material.)

print(“You are \(age) years old”)

Exercise 5

> What is a playground in the Xcode context? How is it useful in Swift development?

Xcode has a built-in feature called Playgrounds. It's an interactive development environment for developers to experiment Swift programming and allows you to see the result of your code in real-time.

🍇 set 1 sources

https://www.apple.com/swift/

https://devmountain.com/blog/what-languages-are-ios-apps-written-in/

https://devmountain.com/blog/how-to-develop-ios-apps-on-windows/

https://devmountain.com/blog/how-to-build-ios-apps/

https://stevenpcurtis.medium.com/your-first-swift-application-without-a-mac- 79598ad839f8

https://www.softwaretestinghelp.com/xcode-tutorial/#What_Is_Xcode

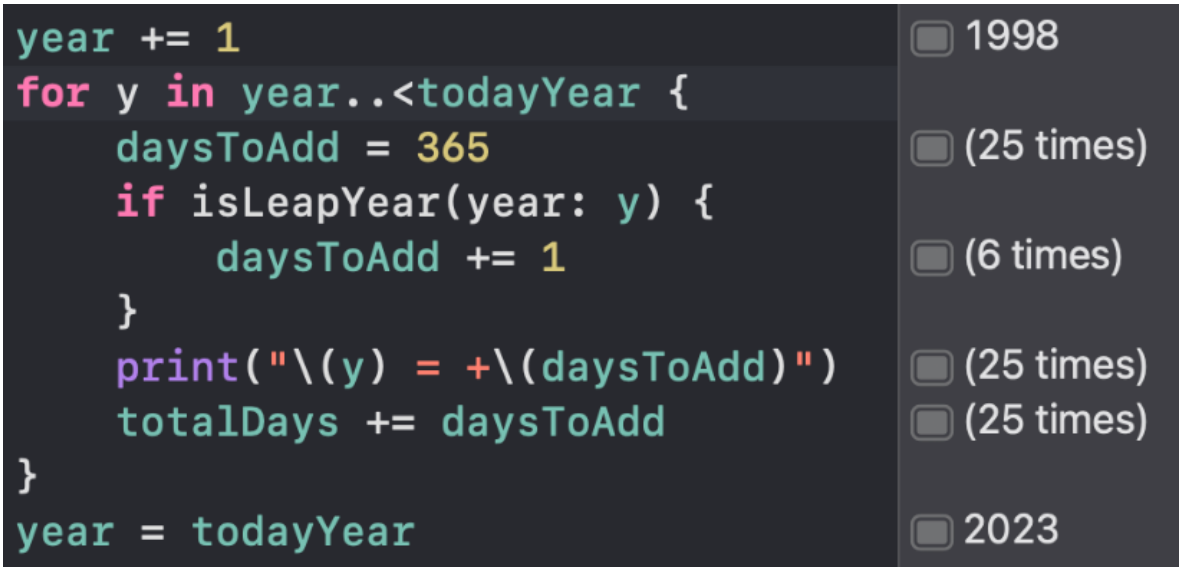
https://www.tutorialspoint.com/explain-the-difference-between-let-and-var-in- swift

Exercise 6

> This is an extract from the “Age” playground from our lab on Friday March 17:

```
// Add the days of all the complete years from the one following the birth year

// to the one before the current year. For normal years, add 365, and for leap

// years, add one more.

year += 1

month = 1

while year < todayYear {

daysToAdd = 365

if isLeapYear(year: year) {

daysToAdd += 1

}

print("\(year) = +\(daysToAdd)")

totalDays += daysToAdd

year += 1

}
```

> It computes the total number of days in the full years which fall between the user’s birth year and the current year.
>
> **TASK**: Rewrite this code using a suitable Swift range and a for loop. See the Ranges playground in Moodle for an example of ranges that we covered in class.



Exercise 7

> This is another extract from the Age playground. It is the function called **dayOfYear(year:month:day)** that calculates the number of the given day in the given year.

```
func dayOfYear(year: Int, month: Int, day: Int) -> Int {

var result = 0

var m = 1

while m < month {

result += daysInMonth(year: year, month: m)

m += 1

}

result += day

return result

}
```

> For example, you could use it to find out that March 20, 2023 is day number 79 of the year.
> When you think about it, you can see that it can be used to calculate the number

> of days in a year if you call it with parameters that represent December 31 in the given year. For example:

```
dayOfYear(year: 2022, month: 12, day: 31)
```

> should result in 365, because 2022 was a normal year. On the other hand, this example:

```
dayOfYear(year: 2020, month: 12, day: 31)
```

> should result in 366, because 2020 was a leap year.
>
> How can you use this information to calculate the days remaining in the user's birth year? Show the Swift source code. Hint: use the difference between the day numbers.

Exercise 8

> Make a helper function out of the age calculation, in the style of **isLeapYear(year:)** and **daysInMonth(year:month:)**. The new function should have the following signature:

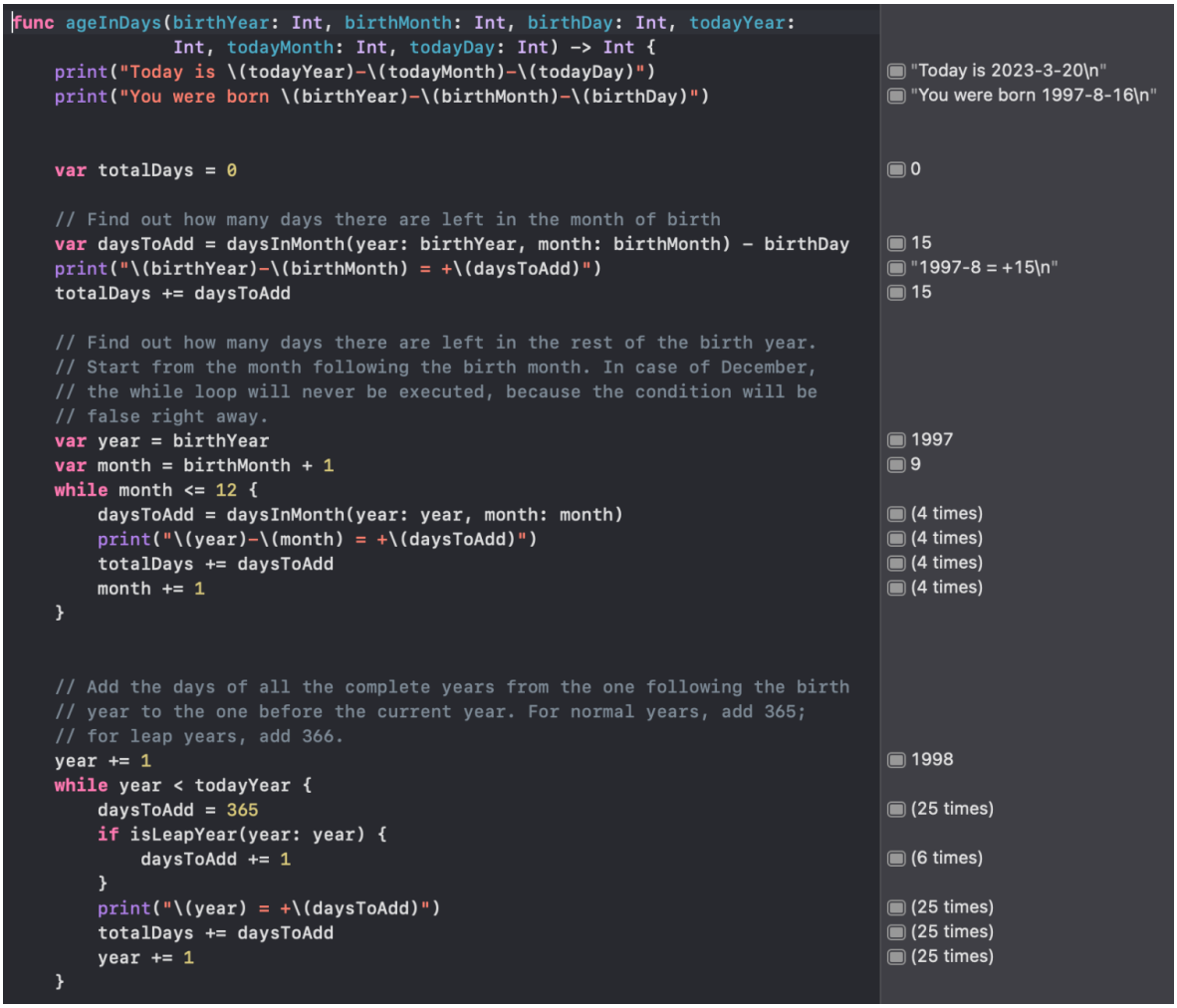
```
func ageInDays(birthYear: Int, birthMonth: Int, birthday: Int, todayYear:
Int, todayMonth: Int, todayDay: Int) -> Int
```

> For a function, six parameters are a lot, so in the future we will be looking at Swift structs and classes, but for now this is fine.

> When you’re done, you should be able to calculate the user’s age like this:

```
ageInDays(birthYear: 1997, birthMonth: 8, birthDay: 16, todayYear: 2023,
todayMonth: 3, todayDay: 20)
```

> Show an example that you tested in a playground.



```
    // Add the days that have passed in the current year, starting from January.
    // First add only the complete months, so go from 1 to one less than the
    // current month.
    month = 1                                                          1
    while month < todayMonth {
        daysToAdd = daysInMonth(year: year, month: month)              (2 times)
        print("\(year)-\(month) = +\(daysToAdd)")                      (2 times)
        totalDays += daysToAdd                                         (2 times)
        month += 1                                                     (2 times)
    }

    // Finally, add the days of the current month up until yesterday.
    daysToAdd = todayDay - 1                                           19
    print("\(year)-\(month) = +\(daysToAdd)")                          "2023-3 = +19\n"
    totalDays += daysToAdd                                             9346
    return totalDays                                                   9346
}

var totalDays = ageInDays(birthYear: 1997, birthMonth: 8, birthDay: 16,    9346
    todayYear: 2023,
todayMonth: 3, todayDay: 20)

    print("Your age in days is \(totalDays)")                          "Your age in days is 9346\n"
```

Exercise 9

> Foundation is a library that contains much of the Swift standard library, including basic data types and functions that are inherited from the early days of macOS (which used to be called OS X).
> You can use all of Foundation by inserting the statement **import Foundation** at the start of your Swift program:

```
import Foundation
```

> One useful data type in Foundation is **Date**, which encapsulates both date and time of day. In this exercise we use Date objects, but we are only interested in the actual date, not the time of day. You can get the current date and time from the operating system.
> Another Foundation data type we need is **Calendar**, which gives us a reference point for the Date objects we use. You can get the current calendar from the operating system and use it to extract date components.

```
// Get the current date from the computer's internal clock

let today = Date()

// Get the current calendar, and use it to get today’s components
```

```
let calendar = Calendar.current

let todayYear = calendar.component(.year, from: today)

let todayMonth = calendar.component(.month, from: today)

let todayDay = calendar.component(.day, from: today)
```

> You can also construct Date objects that have certain date and time components, using another Foundation class called **DateComponents**. For example, someone's birth date could be expressed like this:

```
var birthDateComponents = DateComponents()

birthDateComponents.year = 1997

birthDateComponents.month = 8

birthDateComponents.day = 16

let birthDate = calendar.date(from: birthDateComponents)!
```

> In Swift, functions can be overloaded by their parameters. This means that you can have many functions with the same name, but with different parameters. Based on this information, write a new version of the **ageInDays** function of Exercise 8. It should have the following signature:

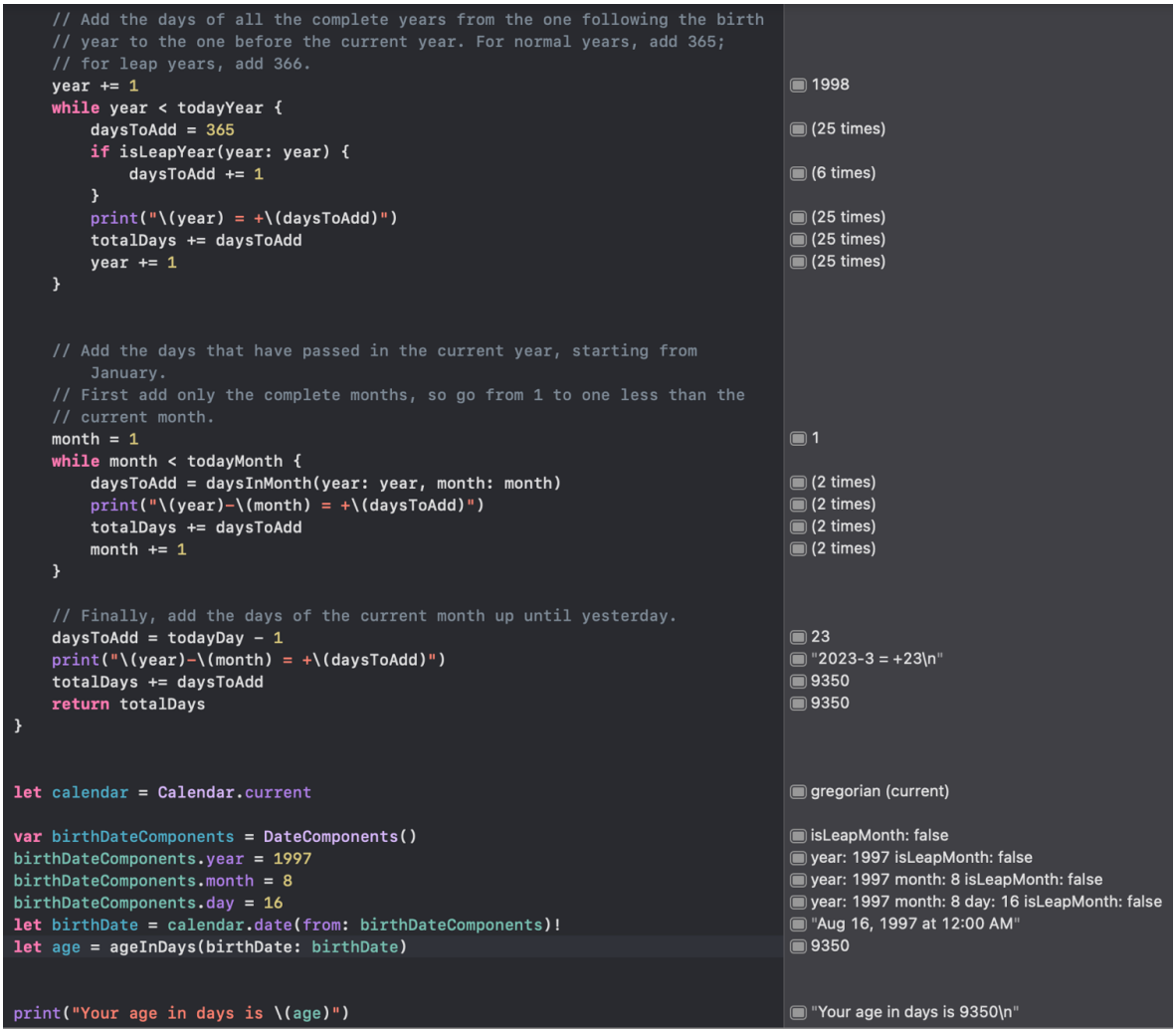
```
func ageInDays(birthDate: Date) -> Int
```

> Make the function calculate the difference between the given birthdate argument and the computer's clock.
>
> When you are finished, you should be able to get the user's age in days simply with this code:

```
var birthDateComponents = DateComponents()

birthDateComponents.year = 1997

birthDateComponents.month = 8

birthDateComponents.day = 16
```

```
let birthDate = calendar.date(from: birthDateComponents)!

let age = ageInDays(birthDate: birthDate)
```

```
func ageInDays(birthDate: Date) -> Int {
    // Get the current date from the computer's internal clock
    let today = Date()                                                  "Mar 24, 2023 at 1:46 AM"
    // Get the current calendar
    let calendar = Calendar.current                                     gregorian (current)
    // use it to get today's components
    let todayYear = calendar.component(.year, from: today)             2023
    let todayMonth = calendar.component(.month, from: today)           3
    let todayDay = calendar.component(.day, from: today)               24
    // use it to get birthday's components
    let birthYear = calendar.component(.year, from: birthDate)         1997
    let birthMonth = calendar.component(.month, from: birthDate)       8
    let birthDay = calendar.component(.day, from: birthDate)           16



    print("Today is \(todayYear)-\(todayMonth)-\(todayDay)")            "Today is 2023-3-24\n"
    print("You were born \(birthYear)-\(birthMonth)-\(birthDay)")       "You were born 1997-8-16\n"


    var totalDays = 0                                                   0

    // Find out how many days there are left in the month of birth
    var daysToAdd = daysInMonth(year: birthYear, month: birthMonth) - birthDay  15
    print("\(birthYear)-\(birthMonth) = +\(daysToAdd)")                 "1997-8 = +15\n"
    totalDays += daysToAdd                                              15

    // Find out how many days there are left in the rest of the birth year.
    // Start from the month following the birth month. In case of December,
    // the while loop will never be executed, because the condition will be
    // false right away.
    var year = birthYear                                                1997
    var month = birthMonth + 1                                          9
    while month <= 12 {
        daysToAdd = daysInMonth(year: year, month: month)               (4 times)
        print("\(year)-\(month) = +\(daysToAdd)")                       (4 times)
        totalDays += daysToAdd                                          (4 times)
        month += 1                                                      (4 times)
    }
```

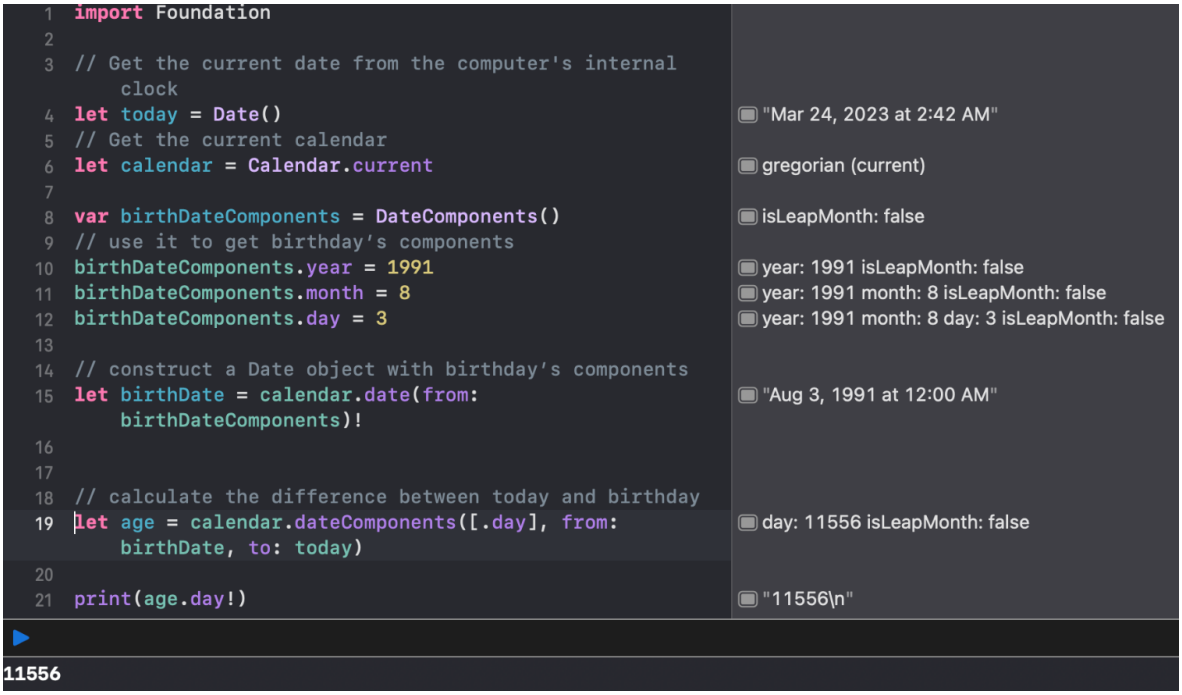


Exercise 10

> Create a new playground and copy the following Swift source code to it. Initialize the fields of **birthDateComponents** with your own birthday.
> Run the code.
> What comes out? Discuss.

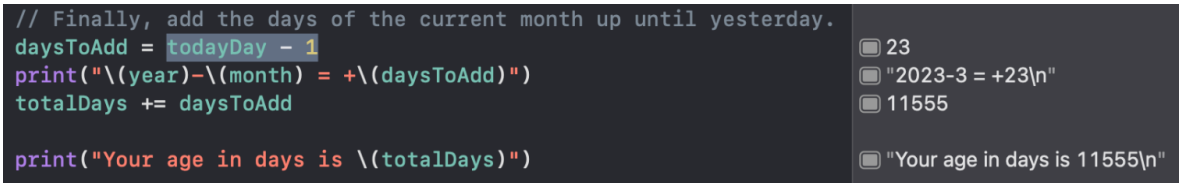
```
let today = Date()

let calendar = Calendar.current

var birthDateComponents = DateComponents()

birthDateComponents.year = 1997

birthDateComponents.month = 8

birthDateComponents.day = 16
```

```
let birthDate = calendar.date(from: birthDateComponents)!

let age = calendar.dateComponents([.day], from: birthDate, to: today)

print(age.day!)
```



The output is my age in days. Steps are explained in the comments above.

I got one day less than here when using the “Age” playground to calculate. Because the Age playground does not count today in.



Here is the usage of **dateComponents():**

dateComponents(_:from:to:)

Returns the difference between two dates specified as `DateComponents`.

iOS 8.0+ | iPadOS 8.0+ | macOS 10.9+ | Mac Catalyst 13.0+ | tvOS 9.0+ | watchOS 2.0+ | Xcode 6.0.1+

Declaration

```
func dateComponents(
    _ components: Set<Calendar.Component>,
    from start: DateComponents,
    to end: DateComponents
) -> DateComponents
```

Parameters

components
Which components to compare.

start
The starting date components.

end
The ending date components.

Return Value

The result of calculating the difference from start to end.

💡 comment from teacher:
Key point: There are many APIs available to ease and speed up our coding. What we need to do is to read the documentation and find out how to use them. The above example “age in days” explains this very well.

🍎 set 2 sources

https://developer.apple.com/documentation/foundation/calendar/2293176-datecomponents

https://sarunw.com/posts/understanding-date-and-datecomponents/

Exercise 11

> What are the Apple Human Interface Guidelines, and why are they needed in iOS development?

Human Interface Guidelines (HIG) is a must-read resource for anyone who wants to master an iOS app design. The information in the HIG (Human Interface Guidelines) is vast and valuable, from how to design your app's interface, navigate content, to manage interactions on iPhone.

By reading HIG, we will gain in-depth information about iOS and essential information about the requirements for the modern iOS app. For example, in HIG, Apple defines the following five basic principles of iOS app design: Consistency, Feedback, Metaphors, Direct manipulation, and User control. The iOS characteristics, including Display, Ergonomics, Inputs, App interactions, System features, help us design a app that users appreciate.

An iOS developer or designer should go through it at least once or twice a year as it is constantly being updated by Apple with the latest design system.

Exercise 12

> In an iPhone app, why is it important to use the standard components available for the user interface?

Build apps quickly from buttons, text labels, and other standard views and controls built into iOS. Customize the appearance of controls or create entirely new views to present our content in unique ways. This approach works well for most apps and shortens development time, since we don't have to spend time creating custom components from scratch.

System-defined components give iPhone users a familiar and consistent experience, which can help increase user trust and satisfaction. Developers should learn how to use and customize those standard components.

> Describe a situation where you would not want to use standard components. Why not?

One situation might be when we need to create a highly customized interface that is tailored to the specific needs of our app, where using standard components may limit the functionality and customization options of our app.

For example, if we are creating an app that is designed to help users manage complex financial data, we may need to use custom components that allow for more advanced data visualization and analysis.

💡 comment from teacher:
data visualization has not traditionally been in the realm of standard user interface components, but interestingly, last year Apple introduced SwiftUI Charts, which also fills this gap.

Exercise 13

> What is an optional type in Swift?

An optional in Swift is basically a constant or variable that can hold a value OR no value. The value can or cannot be nil.

> How do you mark a variable with an optional data type?

It is denoted by appending a “?” after the type declaration. For example:

```
var name: String?
```

The name string is now declared as an optional.

> What does “force unwrap” mean in the context of optional types?

Forced Unwrapping is denoted by “!” to the optional’s name. The exclamation mark says that I know for sure that this optional has a value here, use it. For example:

```
var name: String?
name = "Jingjing Yang"

print(name!)
```

I know for a fact that name has some sort of value in there, thus I used forced unwrapping on the optional variable name.

> Why is it (force unwrap) not always the recommended method of getting the value of an optional variable?

If in reality that the variable does not have a value and you did a force unwrap on it, it will stop the program from running. Thus this method is not always recommended.

```
147  var name: String?
148  //name = "Jingjing Yang"
149  print(name!)
150
```

Line: 157 Col: 1

```
11559
__lldb_expr_34/Age copy.playground:149: Fatal error: Unexpectedly found nil while unwrapping an
Optional value
Playground execution failed:

error: Execution was interrupted, reason: EXC_BREAKPOINT (code=1, subcode=0x18bd55e10).
The process has been left at the point where it was interrupted, use "thread return -x" to return to
the state before expression evaluation.
```

> What other ways are there?

A more graceful method is Optional Binding.

```
147  var name: String?                                  nil
148  //name = "Jingjing Yang"
149
150  if let actualName = name {
151      print("The value is: \(actualName)")
152  } else {
153      print("name is nil")                           "name is nil\n"
154  }
155
```

```
name is nil
```

In the Optional Binding example, the name is checked if it has a value. If it has a value, we will implicitly unwrap and assign name to equal actualName as a temporary constant. This temporary constant can now be used in the conditional block. If name = nil, go to the else statement. This forgoes the need to use the “!” since the value was copied over to the new variable.

Exercise 14

> In the online labs on Friday March 24 we sketched a Swift struct that represents a country in the European Union. Watch the recording of the session to find out more. Here is the definition of the struct

```
struct Country {

var code: String

var name: String
```

```
var area: Int

var population: Int

var populationDensity: Int {

return self.population / self.area

}

}
```

> We also covered arrays and dictionaries. This exercise will use an array type.
>
> Write a function that takes an array of Country objects as a parameter and returns the average population density of all the countries. The signature of the function should be:

```
func getAveragePopulationDensity(countries: [Country]) -> Int
```

```swift
// a struct that represents a country in the European Union
struct Country {
    var code: String // the ISO 3166-1 code for the country
    var name: String // the name of the country (English)
    var area: Int // sq.km
    var population: Int
    var populationDensity: Int {
        return self.population / self.area
    }
}

// This function takes an array of Country objects as a parameter
// and returns the average population density of all the countries.
// using population and area properties of the Country struct
func averagePopulationDensity(countries: [Country]) -> Int {
    var totalPopulation = 0
    var totalArea = 0

    for country in countries {
        totalPopulation += country.population
        totalArea += country.area
    }

    return totalPopulation / totalArea
}

// Make some realistic Country objects
var finland = Country(code: "FI", name: "Finland", area: 338_435, population: 5_553_000)
var sweden = Country(code: "SE", name: "Sweden", area: 438_574, population: 10_481_937)
var estonia = Country(code: "EE", name: "Estonia", area: 45_227, population: 1_331_824)
var germany = Country(code: "DE", name: "Germany", area: 357_340, population: 83_695_430)
var denmark = Country(code: "DK", name: "Denmark", area: 42_921, population: 5_910_577)

// Test the function with a list of countries
var countries = [finland, sweden, estonia, germany, denmark]

print("Average population density of \(countries[0].name), \(countries[1].name),
    \(countries[2].name), \(countries[3].name) 4and \(countries[4].name) is
    \(averagePopulationDensity(countries: countries)) people/sq.km")

```

```
Average population density of Finland, Sweden, Estonia, Germany 4and Denmark is 87 people/sq.km
```

> Test your function with a list of countries.

```swift
// a struct that represents a country in the European Union
struct Country {
    var code: String // the ISO 3166-1 code for the country
    var name: String // the name of the country (English)
    var area: Int // sq.km
    var population: Int
    var populationDensity: Int {
        return self.population / self.area
    }
}

// This function takes an array of Country objects as a parameter
// and returns the average population density of all the countries.
// using population and area properties of the Country struct
func averagePopulationDensity(countries: [Country]) -> Int {
    var totalPopulation = 0
    var totalArea = 0

    for country in countries {
        totalPopulation += country.population
        totalArea += country.area
    }

    return totalPopulation / totalArea
}

// Make some realistic Country objects
var finland = Country(code: "FI", name: "Finland", area: 338_435, population: 5_553_000)
var sweden = Country(code: "SE", name: "Sweden", area: 438_574, population: 10_481_937)
var estonia = Country(code: "EE", name: "Estonia", area: 45_227, population: 1_331_824)
var germany = Country(code: "DE", name: "Germany", area: 357_340, population: 83_695_430)
var denmark = Country(code: "DK", name: "Denmark", area: 42_921, population: 5_910_577)

// Test the function with a list of countries
var countries = [finland, sweden, estonia, germany, denmark]

print("Average population density of \(countries[0].name), \(countries[1].name),
    \(countries[2].name), \(countries[3].name) 4and \(countries[4].name) is
    \(averagePopulationDensity(countries: countries)) people/sq.km")

```

```
Average population density of Finland, Sweden, Estonia, Germany 4and Denmark is 87 people/sq.km
```

Exercise 15

> You are building a SwiftUI view to show information about a country in the European Union. What **SwiftUI controls** would you use for each of the following:

- An image of the flag of the country

Image ----- A view that displays an image.

- The name of the country

TextField ----- A control that displays an editable text interface.

- The area of the country

TextField ----- A control that displays an editable text interface.

- The population of the country

TextField ----- A control that displays an editable text interface.

- A link to the Wikipedia entry about the country

Link ----- A control for navigating to a URL.

- Information about whether the country is a current member or ex-member of the EU

Picker ----- A control for selecting from a set of mutually exclusive values.

> 💡 comment by teacher:
> I don't think Picker would be an appropriate view to use for this kind of data. Also, Toggle is probably more suitable for data that the user can actually change.

- Information about whether the country is a member of the eurozone and/or the Schengen Treaty

Toggle ----- A control that toggles between on and off states.

> 🥥 set 3 sources

https://uxplanet.org/apples-human-interface-guidelines-overview-5d42c2088efc

https://xd.adobe.com/ideas/principles/app-design/ios-app-design/

https://medium.com/@ranleung/understanding-optionals-in-swift-540bfa0e44c7

https://github.com/jerekapyaho/eumemberdata/blob/master/country_name.csv

https://github.com/jerekapyaho/eumemberdata/blob/master/country.csv

https://developer.apple.com/documentation/swiftui

https://www.tutorialspoint.com/ios/ios_ui_elements.htm

https://developer.apple.com/documentation/swiftui/controls-and-indicators?changes=_9

Exercise 16

> Explain the purpose of a binding in SwiftUI, and how and why it is used.

A binding in SwiftUI is a connection between a view and a piece of data. It allows the view to read and write the data, and any changes made to the data will automatically update the view.

Bindings are used to create interactive user interfaces, as they allow the user to manipulate data and see the changes reflected in real time.

Exercise 17

> In SwiftUI the recommended approach is to use many small, reusable views, possibly nested within each other.
> In this context, what does it mean to “extract a subview” in SwiftUI development?
> How can it be useful for you as an application developer?

Extracting a subview in SwiftUI development means taking a part of the view's code and putting it into a separate view, which can be reused in multiple places.

This makes the code more modular and easier to read, maintain, and test. When a view becomes too complex or contains a lot of functionality, it can be beneficial to extract a subview.

Exercise 18

> SwiftUI applications are developed using Xcode. In that context, what is a preview in SwiftUI development?
> How can it be useful for you as an application developer?
> How can you provide data for a preview?

In SwiftUI development, a preview is a live rendering of a view or a view hierarchy that shows how the view will look like in different states and configurations.

Previews are useful for quickly iterating and testing different layouts, styles, and data without running the actual application.

We can use the PreviewProvider protocol to provide data for a preview in SwiftUI. This protocol allows us to create a preview for a view or a view hierarchy using a static previewProvider property that returns a preview. Inside the preview, we can provide data by instantiating the view and passing in the required data using the view's initializer.

Exercise 19

> In the context of developing for Apple platforms, what are SF Symbols, and why and how are they used?

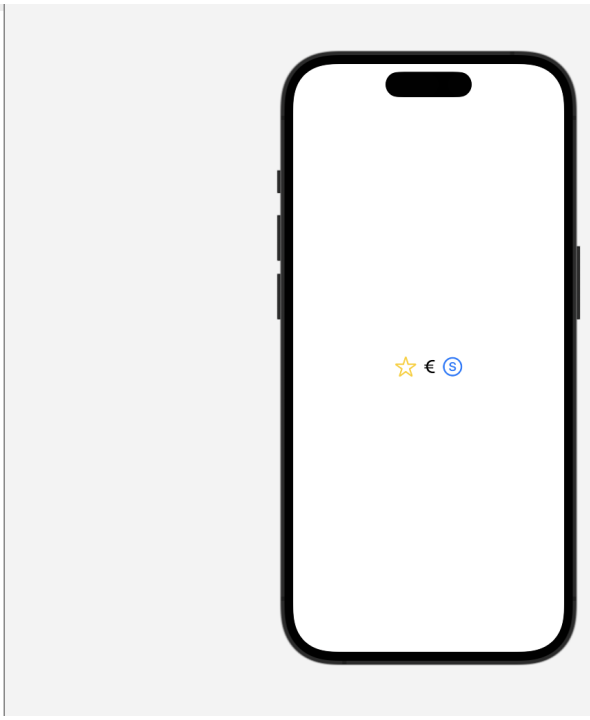
SF Symbols are a set of over 4,400 icons designed to integrate seamlessly with San Francisco, the system font for Apple platforms. They are vector-based and can be resized and colored to fit the needs of the app.

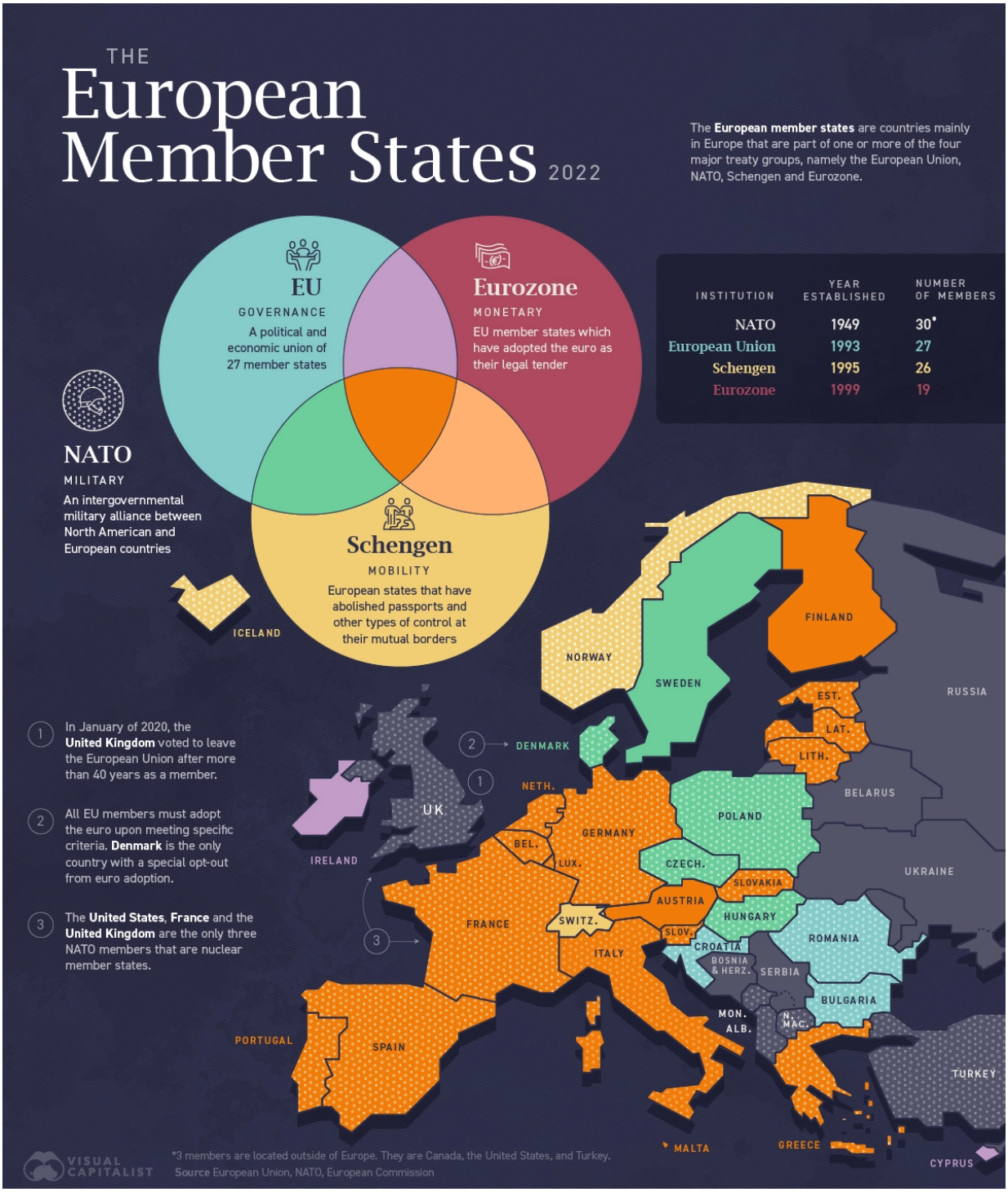
SF Symbols is a feature available starting from iOS 13. In contrast with normal fixed size images, they are made to be used in conjunction with text: their size is specified with a font size, and their baseline can be properly aligned with text.

SF Symbols are used to provide a consistent and recognizable visual language across different applications. Symbols come in nine weights and three scales, and automatically align with text labels. They can be exported and edited using vector graphics editing tools to create custom symbols with shared design characteristics and accessibility features.

> Can you find a suitable symbol for expressing the membership in the Schengen agreement? (The agreement guarantees free movement between participating member countries of the European Union, see Schengen Agreement - Wikipedia.)
> If you find one, try it in a Swift playground or an actual iOS app, using the SwiftUI Image view with the systemName: parameter.

```swift
import SwiftUI

struct MembershipView: View {
    @Binding var country: Country

    var body: some View {
        HStack {
            if country.isMember {
                Image(systemName: "star")
                    .font(.system(size: 28.0))
                    .foregroundColor(.yellow)
            }

            if country.isEuroZone {
                Text("\u{20AC}")
                    .font(.system(size: 28.0))
            }

            Image(systemName: "s.circle")
                .font(.system(size: 28.0))
                .foregroundColor(.blue)
            }
    }
}

struct MembershipView_Previews: PreviewProvider {
    static var previews: some View {
        MembershipView(country: .constant(finland))
    }
}
```





Exercise 20

> In class on Friday April 14 we added some members to the Country struct (the additions are in bold text):

```
struct Country {
var code: String
var name: String
var area: Int
var population: Int
var populationDensity: Int {
return self.population / self.area
}
var isMember: Bool // is the country an E.U. member?
var isEuroZone: Bool // is the country a member of the eurozone?
}
```

> Make a new struct or class called CountryManager, with a property that returns an array of
> Country values. Add an initializer that fills the array with at least six Country values, representing different European Union member countries.
>
> You can find the country data online in this GitHub repository:
> jerekapyaho/eumemberdata: Data and utilities related to the member countries of the European Union (github.com)
> See the files country.csv and membership.csv in the repository for the data. You will need to
> add the new properties to any existing Country initializer calls.
>
> To test your new CountryManager you can use either a Swift playground or an iOS app. This will be useful in the final exercise.

```swift
import Foundation

let austria = Country(code: "AT", name: "Austria", area: 83_879, population: 9_027_999, isMember: true, isEuroZone: true)
let finland = Country(code: "FI", name: "Finland", area: 338_435, population: 5_555_300, isMember: true, isEuroZone: true)
let france = Country(code: "FR", name: "France", area: 632_833, population: 67_897_000, isMember: true, isEuroZone: true)
let germany = Country(code: "DE", name: "Germany", area: 357_430, population: 83_695_430, isMember: true, isEuroZone: true)
let greatBritain = Country(code: "GB", name: "Great Britain", area: 248_528, population: 60_800_000, isMember: false, isEuroZone: false)
let italy = Country(code: "IT", name: "Italy", area: 301_338, population: 60_252_824, isMember: true, isEuroZone: true)
let spain = Country(code: "ES", name: "Spain", area: 505_990, population: 47_100_396, isMember: true, isEuroZone: true)
let sweden = Country(code: "SE", name: "Sweden", area: 450_295, population: 10_379_295, isMember: true, isEuroZone: false)
let denmark = Country(code: "DK", name: "Denmark", area: 42_933, population: 5_814_461, isMember: true, isEuroZone: false)

struct Country {
    var code: String
    var name: String
    var area: Int
    var population: Int
    var populationDensity: Int {
        return self.population / self.area
    }
    var isMember: Bool // is the country an E.U. member?
    var isEuroZone: Bool // is the country a member of the eurozone?

}

struct CountryManager {
    var countries: [Country]

    //    an initializer that fills the array with 6 Country values
    init() {
        self.countries = [austria,finland,france,germany,greatBritain,italy]
    }
}
```

```swift
var countryList = CountryManager()
for country in countryList.countries{
    print(country.name)
}
print("----------------------")

print(countryList.countries[4].code) // the 5th country in the list, which is Germany
print("----------------------")

countryList.countries.append(spain)
countryList.countries.append(sweden)
countryList.countries.append(denmark)
for country in countryList.countries{
    print(country.name)
}
```

```
Austria
Finland
France
Germany
Great Britain
Italy
----------------------
GB
----------------------
Austria
Finland
France
Germany
Great Britain
Italy
Spain
Sweden
Denmark
```

💡 comment from teacher:
You might want to move top-level Definitions of countries to inside the CountryManager initializer, and get rid of the intermediate variables, unless you really want to keep them around for previews.

🥝 set 4 sources

https://www.appypie.com/binding-swiftui-how-to

https://medium.nextlevelswift.com/extract-a-subview-and-preview-f0f817939fe6

https://developer.apple.com/sf-symbols/

https://www.david-smith.org/blog/2023/01/23/design-notes-18/

https://developer.apple.com/documentation/uikit/uiimage/creating_custom_symbol_images_for_your_app

https://techlife.cookpad.com/entry/2021/01/05/custom-symbols-en

Exercise 21

> What are protocols in Swift?

Protocols in Swift are a way of defining a set of methods and properties that a type can adopt. It is a blueprint that can be used to ensure that a class or a struct has certain functionality.

> How do you make a struct or class “conform to a protocol”, or “adopt a protocol”?

To make a struct or a class conform to a protocol, we need to declare its conformance by adding the protocol name after the type name, separated by a colon. Multiple protocols can be listed, and are separated by commas.

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

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

For example, to make a struct conform to the Equatable protocol, we can do:

```
struct MyStruct: Equatable {
    // implementation here
}
```

Exercise 22

> What is the purpose of the Identifiable protocol in SwiftUI?

The Identifiable protocol in SwiftUI is used to provide a stable notion of identity to a class or value type. It requires that a conforming type has a property called id that is unique across all instances of the type.

> Describe a situation where you would need to use it.

We would use Identifiable when we have a collection of data that we want to display in a SwiftUI view, such as a List or a ForEach. The view needs to know which element is which to be able to render them correctly, and Identifiable provides a convenient way to do this.

For example, we could define a `User` type with an `id` property that is stable across our app and our app's database storage. We could use the `id` property to identify a particular user even if other data fields change, such as the user's name.

Exercise 23

Explain property wrappers in Swift.

Property wrappers in Swift are a way of adding functionality to a property. They allow us to define a set of rules or transformations that apply to the property's value in a reusable way.

We use it if we need to perform validation, formatting, or to add extra behavior to a property.

> How are they used in SwiftUI?

Property wrappers are defined using a `@` symbol followed by the name of the wrapper. We can then apply the wrapper to a property by prefixing the property with the wrapper name.

With property wrappers, we can declare a property with an initial value, and the property wrapper will automatically generate the necessary code to manage the state of that property. This makes it easier to manage the state of a view and ensures that the view is always up-to-date with the latest data.

> Give an example.

The most common use case for property wrappers in SwiftUI is the @State property wrapper, which allows a view to manage its own state. When the value of a @State property changes, SwiftUI automatically re-renders the view to reflect the new value.

```
@State private var username = ""
```

In addition to @State, there are several other property wrappers in SwiftUI, such as @Binding, @ObservedObject, and @EnvironmentObject. These property wrappers are used to manage more complex data structures and to share data between views.

Exercise 24

> What does navigation mean in an iOS application?

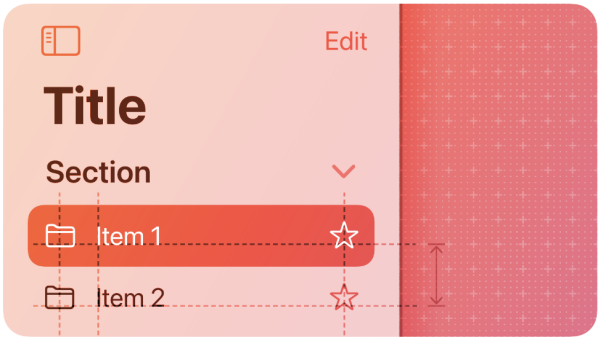
Navigation in an iOS application refers to the ability to move between different screens or views within the app. It is a fundamental part of the user experience, as it allows the user to explore and interact with the app's content.

> What are the most common navigation patterns in iOS, and how are they used in SwiftUI? Looking at Apple’s Human Interface Guidelines may give you ideas.

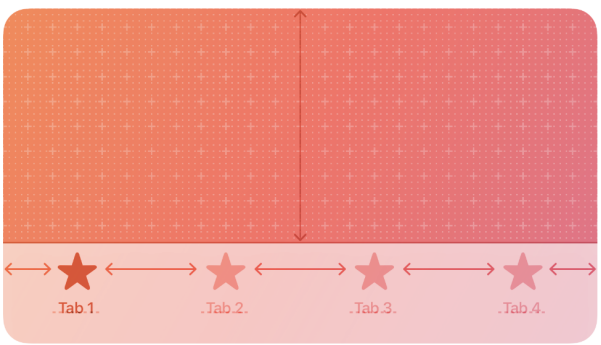
The most common navigation patterns in iOS are:

Navigation Bar: A bar at the top of the screen that displays the current view's title and provides navigation buttons, such as a Back button to return to the previous view.

Sidebar: A sidebar enables app navigation and provides quick access to top-level collections of content in your app or game. It is typically accessed by swiping from the left edge of the screen or tapping on a menu icon.



Tab Bar: A bar at the bottom of the screen that allows users to switch between different sections of an app by tapping on different tabs . Each tab typically represents a different view or screen within the app.



In SwiftUI, navigation is handled by the NavigationView and NavigationLink views. NavigationView provides a navigation bar and a space to display the current view's content, while NavigationLink allows the user to navigate to another view by tapping on it.

Exercise 25

> In class on Friday April 21 we made a class called CountryManager, with one property called countries that had the @Published property wrapper. (See the file EUMembers_Final_2023-04-21.zip in Moodle for the class definition.)
>
> What is the purpose of this property wrapper for SwiftUI apps? (Hint: look up “single source of truth” in SwiftUI.)

The @Published property wrapper in SwiftUI is used to create a binding between a property and a view. It allows us to create a "single source of truth" for our data, so that any changes made to the data are automatically reflected in the view.

> How is it useful in the app we are developing?

It is useful in the app we are developing because it allows us to store the list of countries in a centralized location (the CountryManager class), and make it available to all the views in the app. When we add or remove countries from the list, the changes are automatically propagated to all the views that display the list.

set 5 sources
https://docs.swift.org/swift-book/documentation/the-swift-programming-language/protocols/
https://developer.apple.com/documentation/swift/identifiable
https://www.swiftbysundell.com/articles/property-wrappers-in-swift/
https://uxplanet.org/ios-navigation-styles-and-which-one-should-we-choose-in-apps-52d06524254e
https://developer.apple.com/design/human-interface-guidelines/components/navigation-and-search/navigation-bars/
https://www.hackingwithswift.com/articles/216/complete-guide-to-navigationview-in-swiftui
https://developer.apple.com/documentation/swiftui/navigationview
https://developer.apple.com/documentation/swiftui/navigationlink
https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-published-property-wrapper