

Some SwiftUI fundamentals

[[PDF version](#) unfortunately with some spacing and page splitting problems]

[Guy Lapalme](#), Université de Montréal, October 2023

This document was originally written to help me remember what I consider to be the [SwiftUI](#) fundamentals. Finally, it became the tutorial that I would have like to read before working with SwiftUI. I hope it is as useful to others as it has been for me writing it.

Many documents on the web, notably [Apple Tutorials](#), usually demonstrate SwiftUI features by means of examples without never taking time to explain the underlying ideas. If it can be relatively easy, but not always, to build spectacular demos by following the steps of a tutorial, it is not so straightforward to apply these ideas later in one's own application. There are also many commercial or ad-full web sites and videos that explain some ideas, but there are usually limited to a few minutes, [even seconds](#), reading focused on aspects of the language and system ¹.

A notable exception to this superficial approach was the first edition of [Thinking in Swift](#), which aimed to explain the rationale behind SwiftUI, but even there I found it a bit hard to follow. So I was looking forward for their second edition. Unfortunately, this new edition is even less explicit on the fundamentals than their first one. The authors go to great length to illustrate some aspects, but they skip interesting explanations about the flow of information in SwiftUI that appeared in the first edition. They seem to take for granted that the reader is a seasoned Swift programmer at ease with more advanced features of the language or that they have followed their on-line courses. The main advantage of the second edition is that it explains important changes in the state management that appeared in iOS 17.

[Swift](#) is a script-like language combining great insights from years of programming language design. I see it as a Python-like script language with static strong typing *à la* Haskell. Its designers have also introduced some *uncommon* features, such as *result builders* or *property wrappers* that are heavily used in SwiftUI ². This is why I will spend some time explaining these in this document.

I take it for granted that the reader has already some basic notions of the Swift language and some experience working with Xcode. [Link to the Xcode project of the examples in this document](#). At the beginning of some sections of this document, links to the complete source code for that section are also given.

1. Main principle: linking states and views

Interactive applications must ensure that the display always reflects the states, *sources of truth* in the Apple of terminology, that define the system. When the user or an external event modifies states, the display must be updated to reflect these changes. Many frameworks have been developed over the years to help build systems for synchronizing the states and display. The most well known being the *Model-View-Controller* approach in which the controller updates the view when notified of model changes, usually by means of callback functions.

Recently, a new paradigm has appeared which bypasses the controller, so that the system is defined in terms of a view which is updated *automatically* when the model changes. One well-known example being [React](#). See [this article, for a comparison between SwiftUI and React](#).

According to this [Apple introductory document](#), you create a lightweight description of your user interface by declaring views in a hierarchy that mirrors the desired layout of your interface. SwiftUI then manages drawing and updating these views in response to events like user input or state changes.

[According to Bart Jacobs](#), SwiftUI provides developers with an API to declare or describe what the user interface should look like. SwiftUI inspects the declaration or description of the user interface and converts it to your application's user interface. A view is no longer the result of a sequence of events. A view is a function of state. In other words, the user interface is a derivative of the application's state. If the application's state changes, the user interface automatically updates, eliminating a whole category of bugs.

To explain how to achieve these laudable goals, I will start by explaining how to build a view from a hierarchy of other views. When developing in SwiftUI with Xcode, views are defined by Swift code which can be executed on the fly in a preview window, so that it is easy to see how it is laid out. Xcode allows building a SwiftUI view by drag-and-drop of code snippets and patterns, but finally it seems simpler to me to type the code directly. Once the views can be defined, I will describe how to synchronize them with system states.

2. View

`View` is a predefined Swift type, a *protocol*, the equivalent to an *interface* in Java. Remember that SwiftUI `View` instances, being `struct`, are **immutable**, so they must be recreated when some of its content changes. SwiftUI provides a whole gamut of predefined views that can be combined to form new custom views.

2.1. Predefined views

- **Controls**: basic interface elements such as `Text`, `Button` or `Label`, but also more sophisticated such as `DatePicker`, `List`, `Table` or `TextEditor`
- **Layout**: organizers such as `HStack`, `VStack`, `ZStack` or `Spacer`
- **Other**: such as `Circle`, `Divider`, `Group` or `Image`
- **View modifiers**: return a modified version of the view by means of the *dot notation* for different purposes, for example
 - changing its look: `.font()`, `.padding()`, `.border()`
 - changing its behavior: `.onChange{}`, `.onAppear{}`, `.onTapGesture{}`

2.2. Custom views

A new view is defined with a `struct` declaration that must conform to the `View` protocol which means that there must be a `body` property of the type `some View` of the following form.

```
1 struct aCustomView: View {
2     var body: some View {
3         .. content of the view ..
4     }
5 }
```

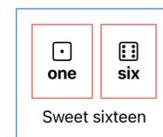
2.3. Show two dice

[\[source code\]](#)

We begin with an application that displays views without any interaction. The left part of following figure is an example of custom view (`ImageNameView`) that is used twice within the scaffolding code provided by Xcode with the corresponding preview. It illustrates the following interesting features:

- line 11: definition of a custom view with the following components:
 - `image`: mandatory string parameter, because of `let`, corresponding to the name of a system image (found in the Xcode Symbols library)
 - `name`: optional string parameter, because of `var` with initialization, the text to display
 - `body`: defining how the View is composed of other views: here the `Image` view is stacked vertically on top of a `Text`. These views are slightly modified, the `Image` is made larger and the `Text` set in bold. The whole stack is then added some space (padding) around it and a red border is applied.
- line 24: `ContentView` (this name is conventional, but it should match the one used in `ContentView_Previews` and in the `body` of the app when run on a device) whose `body` defines the view that will be displayed when run on the device. In this case, this is a *Horizontal Stack* of two calls to the custom view I have previously defined each with a different parameter. This stack is then itself stacked vertically with a text below with some padding and a blue border.
- line 37: `ContentView_Previews`: (provided by the Xcode scaffolding) defines what will be displayed in the simulated device on the right. In this case, it is merely the `ContentView`. This allows following the how the global view develops like while coding it. This is especially useful when learning how to layout views in SwiftUI.

```
10 // custom view for an image over a text with a red border with padding
11 struct ImageNameView:View {
12     let image:String // mandatory parameter
13     var name:String = "one" // optional parameter
14     var body: some View {
15         VStack {
16             Image(systemName:image).imageScale(.large)
17             Text(name).fontWeight(.bold)
18         }.padding()
19         .border(.red)
20     }
21 }
22
23 // use the custom view multiple times
24 struct ContentView: View {
25     var body: some View {
26         VStack {
27             HStack {
28                 ImageNameView(image: "die.face.1")
29                 ImageNameView(image: "die.face.6", name: "six")
30             }
31             Text("Sweet sixteen")
32         }.padding().border(.blue)
33     }
34 }
35
36 // use the ContentView for the Preview (useful for designing)
37 struct ContentView_Previews: PreviewProvider {
38     static var previews: some View {
39         ContentView()
40     }
41 }
```



The syntax used for defining the value of the body of a View is defined by a specialized *result builder* which is a predefined *property wrapper*. These concepts are interesting in themselves, so I spend some time explaining them independently of SwiftUI. Views also use *trailing closures*, a notation which might be unfamiliar to programmers in other languages than Swift. I will first present this notation used throughout Swift programs.

3. Trailing closure

Swift uses *closure* to designate an anonymous function, often called *lambda* in other programming languages. A closure is a list of expressions and instructions within curly brackets. When parameters are specified, they are indicated at the beginning followed by the keyword `in` such as in the following example for computing the sum of squares of two integers and returning an integer. A closure containing a single expression does not need a `return` keyword in front.

```
1 { (a:Int, b:Int)->Int in a*a + b*b }
```

In most contexts, type annotation of parameters and result can be inferred by the Swift compiler, such as in the following expression returning 25

```
1 {a,b in a*a + b*b}(3,4) // => 25
```

A closure is most often used as a functional parameter for functions like `map` that transforms all elements of an array to create a new one in with elements of the original list modified by the function. For example, the next expression creates a list of corresponding squares of two lists of integers. It uses `zip` to build a list of pairs from corresponding elements of two lists.

```
1 zip([1,2,3],[4,5,6]).map({a,b in a*a + b*b}) // => [17,29,45]
```

As the closure is the last (and only) functional parameter of `map`, the call can also be written as:

```
1 zip([1,2,3],[4,5,6]).map {  
2   a,b in a*a + b*b  
3 } // => [17,29,45]
```

Here I use several lines to show how it would appear if the content of the function would be more elaborated. The fact that it appears without the enclosing parentheses is called a *trailing closure*.

There are also some other *goodies* for simplifying closures (not only trailing ones) especially in the case of single expressions. Parameters can be referenced implicitly by position with a dollar sign. The above expression could thus be written as:

```
1 zip([1,2,3],[4,5,6]).map {$0*$0 + $1*$1} // => [17,29,45]
```

And even better (or worse !), a closure of the form `{ $0 op $1 }`, where `op` is a binary operator, can be simplified as `op`, but then it must be used as a parameter not a trailing closure. For example,

```
1 zip([1,2,3],[4,5,6]).map(+) // => [5,7,9]
2 (1...5).reduce(1,*) // => 125 i.e. 5! "reduce" is often called "foldRight"
```

4. Property wrapper

[\[source code\]](#)

A property wrapper is a notation that encapsulates read and write access to a value and adds additional behavior to it. A Swift property is the name for accessing a value in an object, in SwiftUI most often a `struct`. A property can be either *stored* (as in most programming languages) or *computed* which means that the value is recomputed at each time its value is needed with a *getter* function or changed with a *setter* function.

I start with a very simple example of a `struct` for a die whose values must be between 1 and 6. Internally, an instance of `Die` uses a value between 0 and 5 but this fact is hidden from the user. The *concrete* value used for computation is the private integer `number` and the private function `limit` converts the value to the acceptable range. `Die` wraps an integer to limit its values between 1 and 6 by taking the value minus one modulo 6 and projects its current value as a string. The value is accessed and modified by the `get` and `set` function of the computed property `wrappedValue`. The `projectedValue` has only a `get`.

This seemingly convoluted terminology for such a simple application will prove to be useful later. Note that any action could be added to the *setter* code here, such as accessing a database, validating a data or refreshing a view!

```
1 struct Die {
2     private var number:Int!// implicitly unwrapped optional to allow limit call in
   init()
3
4     init(wrappedValue:Int){
5         number = limit(wrappedValue)
6     }
7
8     var wrappedValue: Int {
9         get { number+1 }
10        set { number = limit(newValue) }
11    }
12
13    var projectedValue:String {
14        get { ["one", "two", "three", "four", "five", "six"][number] }
15    }
16
17    private func limit(_ val:Int) -> Int {
18        return max(0, (val-1)%6)
```

```
19     }
20 }
```

Given this definition, the creation of an instance of this `struct` would be

```
var d = Die(wrappedValue:10)
```

Accessing the constrained values would be `d.wrappedValue`, in this case 4, or `d.projectedValue`, in this case "four".

This seems a bit cumbersome, but once this `struct` is prefixed with `@propertyWrapper`, the Swift compiler greatly simplifies the access and manipulation because the *wrapped* integer can now be used like any other integer. So the structure definition would be

```
1 @propertyWrapper struct Die {.. same as above ..}
```

The declaration and initialization of an integer of this type become

```
@Die var d = 10
```

The ampersand before a property wrapper creates an *attribute* in the Swift terminology, in a way like *annotations* in other programming languages. Now `d`, whose value is 4, can be used like any integer in an expression such as `d*3+d`, the range constraint being applied transparently. The projected value is obtained with `$d` which returns "four".

So that the compiler knows which variable are *wrapped* and *projected*, **the variable names `wrappedValue` and `projectedValue` must be used in the definition of the `struct`**. As an added bonus, `_v` gives access to the `struct` instance itself, although this is seldom needed.

5. Result builder

[\[source code\]](#)

A [result builder](#) is a user-defined type attribute that adds syntax for creating nested data, like a list or tree, in a natural, declarative way. The code that uses the result builder can include ordinary Swift syntax, like `if` and `for`, to handle conditional or repeated pieces of data. This notation allows the creation of Domain Specific Languages (DSL), see [some spectacular examples](#). It explains how the body of a SwiftUI view can be considered as a single expression without spurious parentheses, brackets and commas.

I now give an example of a result builder, independently of SwiftUI. It is a command-line application featuring a *Blocks world* with the following behavior:

- A `Block` is created from a text (a `String`) with, optionally, a border (a `String` with a single character), a *width*, and a *height*. If the text contains newlines, all lines are centered in the block as for a SwiftUI `Button`. If a border is specified, it is added around the block and if `height` or `width` are specified, they are used for centering the text vertically or horizontally.
- A `Block` can be printed using the method `print()`. Here is a first interaction.

```
1 Block("We\nlove\nSwiftUI", ".").print()
```

which outputs

```
1 .....  
2 . We .  
3 . love .  
4 .SwiftUI.  
5 .....
```

- A `Block` or list of `Block`s can be added to the left, to the right, to the top and to the bottom of another one to create a new `Block`. It is also possible to create copies of a block either horizontally or vertically. Blocks of different height or width are centered relative to one another. A border can be added to the final result.

Here is an example of chained calls with the result,

```
1 Block("We\nlove\nSwiftUI")  
2   .add(right:Block("|").repeating(vertical: 4))  
3   .add(right:[Block("truly"),Block("so", " ")])  
4   .add(bottom:Block("very much", "~",width:10))  
5   .border("+").print()
```

which outputs

```
1 ++++++  
2 + We |      +  
3 + love |truly so +  
4 +SwiftUI|      +  
5 +      |      +  
6 + ~~~~~~      +  
7 + ~very much ~  +  
8 + ~~~~~~      +  
9 ++++++
```

Using the result builder explained later this intricate chain of embedded calls can be simplified with the following call that is more intuitive and reminiscent of a SwiftUI view.

```

1 Vert("+") {
2     Horiz {
3         Block("We\nlove\nSwiftUI")
4         Vert {for _ in 0..<4 {Block("|")}}
5         Block("truly")
6         Block("so", " ")
7     }
8     Block("very much", "~", width:10)
9 }.print()

```

A `resultBuilder` is a predefined property wrapper for building at compile time elaborated data structures or lists of calls. It can be considered as a kind of structured macro system that separates the various components and combines them into calls to the appropriate data structure.

The first thing to define is a `resultBuilder` structure to deal with a list of statements, conditionals and loops. In our case, all functions of the result builder create a list of `Block`s of various forms: variadic parameters in the first case and list of list of blocks in the second and last cases. `buildEither` calls deal with conditional statements and `buildExpression` is applied automatically by the `resultBuilder` interpreter when a single `Block` is encountered. Definitions for `buildEither` (lines 13-18) and `buildArray` (lines 19-21) are optional, but when they are not defined then no conditional statement, nor loop can be used in the calls.

```

1 @resultBuilder struct BlocksBuilder {
2     static func buildBlock(_ components: Block...) -> [Block] {
3         components
4     }
5     // deal also with variadic parameter of list of blocks
6     static func buildBlock(_ components: [Block]...) -> [Block] {
7         Array(components.joined())
8     }
9     // transform a single block into a list of blocks
10    static func buildExpression(_ expression: Block) -> [Block] {
11        [expression]
12    }
13    static func buildEither(first component: [Block]) -> [Block] {
14        component
15    }
16    static func buildEither(second component: [Block]) -> [Block] {
17        component
18    }
19    static func buildArray(_ components: [[Block]]) -> [Block] {
20        Array(components.joined())
21    }
22 }

```

The resulting property wrapper `BlocksBuilder` can then be used to define the following two functions to create an appropriate list of calls to `Block`s. The last parameter is a Swift *trailing closure* that is called to create the list of blocks that are stacked either to the right for `Horiz` or to the bottom for `Vert`. The first parameter is the optional border that is applied.

```

1 func Horiz(_ border:String="",@BlocksBuilder content:() -> [Block])->Block {
2     let blocks = content()
3     return blocks[0].add(right:Array(blocks[1...])).border(border)
4 }
5
6 func Vert(_ border:String="",@BlocksBuilder content:() -> [Block]) -> Block {
7     let blocks = content()
8     return blocks[0].add(bottom:Array(blocks[1...])).border(border)
9 }

```

With these can now write functions to print blocks of numbers, here C_k^n in a pyramid such as this

```

1 Vert{
2     for n in 0...5 {
3         Horiz {
4             for k in 0...n {
5                 Block(String(C(n,k)),width:4)
6             }
7         }
8     }
9 }.print()

```

to create a [Pascal's triangle](#)

```

1         1
2        1  1
3       1  2  1
4      1  3  3  1
5     1  4  6  4  1
6    1  5 10 10  5  1

```

or to print a checkerboard

```

1 let black = Block("*", "*") // create a 3x3 block of 9 "*"
2 let white = Block(" ", " ") // create a 3x3 block of 9 " "
3 Vert("+"){
4     for i in 1..<8 {
5         Horiz {
6             if i%2 == 0 {
7                 for _ in 1...4 {black ; white}
8             } else {
9                 for _ in 1...4 {white ; black}
10            }
11        }
12    }
13 }.print()

```

whose output starts with

```

1 ++++++
2 +   ***   ***   ***   ****+
3 +   ***   ***   ***   ****+
4 +   ***   ***   ***   ****+
5 +***   ***   ***   ***   +
6 +***   ***   ***   ***   +
7 +***   ***   ***   ***   +
8 +   ***   ***   ***   ****+
9 +   ***   ***   ***   ****+

```

6. Back to SwiftUI

The `resultBuilder` for creating views in SwiftUI is (appropriately) called `viewBuilder` which is seldom called directly. It was defined by SwiftUI designers for declaring functions that create the views to perform layout e.g. `VStack`, `HStack`, `List`, etc. The trailing closure explains the peculiar syntax which also relies on the fact that a function definition containing a single expression does not need to add the `return` keyword.

Equipped with this specialized result builder, static views can be built, but the key point is how to link this view with an application model. SwiftUI uses again *property wrappers* which proved to be well adapted for transparently linking view updates with object modifications. Swift being statically typed, SwiftUI designers managed to use types to limit the number of views that need to be recreated when certain variables and objects are changed.

7. State management system

In SwiftUI, a view is bound to some data (or state) as a source of truth, and automatically updates whenever the state changes. Every view has a state, which can be changed during execution and each time the state is changed, the view is **recreated**, remember that a view is immutable.

As shown above, annotating a variable as a `@State` allows executing code at each access and or

As shown above, annotating a variable as a `PropertyWrapper` allows executing code at each access and/or modification. SwiftUI uses this feature to ensure that when a state variable is modified, the view is recreated accordingly to reflect the new value. The code that Swift executes on getter and setter of the state property wrapper is hidden from the user.

SwiftUI provides [17 property wrappers](#), but I present only the five ones that I consider to be fundamental:

`@State`, `@Binding`, `@StateObject`, `@ObservedObject`, `@Environment`.

I first describe the first two attributes that are used for local modifications of simple values and use them in a small application. The other three will be used in the next application.

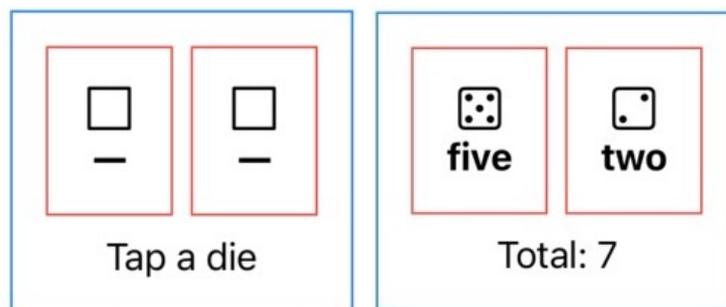
- `@State` is a source of truth for the view and is used when the scope of changes is limited to the current view. The framework allocates a persistent storage for this value type and makes it a dependency, so changes to the state will automatically be reflected in the view. Usually a `@State` annotates a *private* variable, because it is used by the view internally.
- `@Binding` can read and write a value owned by a source of truth, for example, a `@State`. An interesting feature is the fact that the **projected value of a State is a binding** that can be passed in a view hierarchy and change it. The changes will be reflected on any view which uses it as a source of truth.

In iOS 17 and macOS 14, the state management API of SwiftUI has changed while still staying compatible with previous versions. As the basic principles stay the same (specifically `@State` and `@Binding`), we will first describe the *previous* API and [explain later](#) how the same application can be somewhat *simplified* with the new API.

8. Roll two dice

[\[source code\]](#)

The following example, a variation on our [first example](#), shows typical use cases for these two property wrappers. It displays two dice that can be *rolled* by tapping (or clicking) on their view. The global view displays the total of the two dice as shown in the next figure. The left part shows the initial display and the right part a typical display once the user has tapped (or clicked) at least once on each die.



A `DieView` defines `number` as a `@State` corresponding to the number of dots on the last draw. Conceptually, this could be a simple integer value, but as **Swift views are immutable**, `number` must be annotated as a `@State` which performs the appropriate delegation and linking with the view so that when `number` is changed, the view is recreated appropriately. `number` is initially set to 0 and modified in the code associated with the view modifier `.onTapGesture` in which it is set to a random integer between 1 and 6. Updating the total is done by subtracting the current value to get the value of the other die before adding the new value. The `Image`

done by subtracting the current value to get the value of the other die before adding the new value. The `Image` and the `Text` displayed in the view depend on the value of `number`, so each time a new number is regenerated, the `Image` and the `Text` are recreated,

```
1 struct DieView:View {
2     @State private var number:Int = 0
3     @Binding var total:Int
4     var body: some View {
5         VStack {
6             Image(systemName: number==0 ? "squareshape"
7                 : ("die.face."+String(number)))
8                 .imageScale(.large)
9             Text(["-", "one", "two", "three", "four", "five", "six"][number])
10                .fontWeight(.bold)
11        }.padding()
12        .border(.red)
13        .onTapGesture {
14            let other = total - number
15            number = Int.random(in:1...6)
16            total = other + number
17        }
18    }
19 }
```

The `ContentView` stacks two `DieView`s side by side over a text that displays the `total` of the two dice when it is not 0. `total` is a local `@State` to the `ContentView`, but it needs to be passed to the two `DieView`s to be updated. For this, the `Binding` associated with the `State` must be retrieved and sent to each `DieView`. As the `projectedValue` of a `State` is a `Binding` (how convenient!), the binding `$total` is given to each call to `DieView` and any modification done to `total` in a `DieView` will be reflected in the `ContentView` to recreate its `Text` with the new value.

```
1 struct ContentView: View {
2     @State private var total:Int = 0
3     var body: some View {
4         return VStack {
5             HStack{
6                 DieView(total: $total)
7                 DieView(total: $total)
8             }
9             Text(total == 0 ? "Tap a die" : "Total: \(total)")
10        }.padding().border(.blue)
11    }
12 }
```

So conceptually a variable with attribute `@State` is used for storing a local value that is *watched* by the system

So conceptually a variable with attribute `@State` is used for storing a local value that is watched by the system to ensure that the display is updated when it is modified. A variable with attribute `@Binding` has similar properties but it is linked to a `@State` variable in another context. The Swift implementation ensure that the new view (re)creation process is transparent to the user and efficient. This how the *controller* of the Model-View-Controller is bypassed, thus simplifying the flow of information, because when the state is changed, the view is automatically *changed* to reflect the new state.

9. Solitaire Yahtzee

[\[source code\]](#)

To show use cases of other property wrappers available in SwiftUI, I now present an application for playing a solitaire version of the game of [Yahtzee](#). The objective of the game is to score points by rolling five dice to make certain combinations. The dice are rolled three times in a turn to try to make various scoring combinations. On the second or third turn, the player can *fix* some dice and roll only the others. After the third roll, the player chooses a scoring category to add to the total score. Once a category has been selected in the game, it cannot be used again. The scoring categories have varying point values, some of which are fixed values and others for which the score depends on the value of the dice. The goal is to score the most points over the 13 turns.

9.1. Description of the Yahtzee interface

The figure shows three sequential screenshots of the Yahtzee interface. Each screenshot displays five dice at the top and a list of scoring categories below. Buttons for 'Roll' and 'Choose score' are visible.

Category	Turn 1	Turn 2	Turn 3
Aces	—	1	—
Twos	—	2	—
Threes	—	6	—
Fours	—	4	—
Fives	—	0	—
Sixes	—	0	—
Upper section total	0	0	0
Upper section bonus	0	0	0
3 of a kind	—	0	—
4 of a kind	—	0	—
Full house	—	0	—
Small straight	—	30	30
Large straight	—	0	—
Yahtzee	—	0	—
Chance	—	13	—
Lower section total	0	0	30
Grand total	0	0	30

As shown in this figure showing the first turn, the top part is a display of five dice, followed by buttons `Roll` and `C`. Below are shown two sections of possible dice combinations, called categories, with subtotal scores. When tapped, the `Roll` button assign new values to the dice.

The second part on the left of the figure shows the situation once the user has rolled the dice three times. As the three dice in the middle have been *fixed* by clicking on them, thus changing their color, only the values of the first and last dice have changed. The system displays all possible scores: a *Small straight* (a list of 4 consecutive values here 1,2,3 and 4) worth 30 points or 2 *Threes* worth 6 points, an Ace, a Two or Chance which is the sum of all dice values.

In the third part of the figure the user has chosen the *Small straight* by tapping on it and its total is added to

In the third part of the figure, the user has chosen the *small straight* by tapping on it and its total is added to *Lower section total* and *Grand total*. Once a category has been selected, it cannot be selected again. The game continues until all 13 categories have been selected. If at a turn no *paying* category occurs, one must be chosen anyway which results in a 0 score for this turn.

The last part of the figure shows what happens when the user clicks on the `C` button for changing the color of the selection. This change of color would also affect *fixed* dice.

I do not claim that this user interface is the best one or the most intuitive for this game, but it illustrates the use of SwiftUI complex object attributes in a restricted setting. It involves `class` instances, and not only `structs`, with methods and properties, these instances being shared between many views.

I first define the roles of three attributes, although the nuances might seem a bit cryptic at this point, I hope that they will become clearer once they are seen in action.

- `@ObservedObject` This property wrapper annotates a complex object in which it is the user that is responsible for indicating that a part of the object has been changed by calling the `objectWillChange` method required by the `ObservableObject` protocol. This seems complex, but thanks to the property wrapper `@Published` (yet another attribute!), this method is called automatically when the annotated part of the object is changed.
- `@StateObject` is a special kind of `@ObservedObject` that should be used within the view in which the object is created to indicate to SwiftUI it is the owner of this object. Other views that reference this object should use `@ObservedObject`.
- `@EnvironmentObject` This property wrapper also doesn't create or allocate the object itself. Instead, it provides a mechanism to share values across a hierarchy of views.

9.2. Organization of the application

The main view (`ContentView` in the usual SwiftUI terminology) is organized as follows from top to bottom

- View with five dice and two buttons
- Upper section with 6 categories
- Total and bonus of upper section
- Lower section with 7 categories
- Lower section total and grand total

This is coded as follows in the body of the main `ContentView`. The other Views will be explained later.

```
1  var body: some View {
2      VStack {
3          HStack (spacing:3){ // top view with dice and buttons
4              DiceView(dice: dice)
5              Button (move,action:roll_dice)
6                  .frame(width: 90)
7
9              .buttonStyle(.bordered)
```

```

8         .fixedSize()
9         Button ("C", action: selectColor.nextColor)
10    }
11    SectionView(section: upper_section,action: update_totals)
12    CategoryView(kind: "Upper section total",score: $upper_total)
13    CategoryView(kind: "Upper section bonus",score: $upper_bonus)
14    SectionView(section: lower_section,action: update_totals)
15    CategoryView(kind: "Lower section total", score: $lower_total)
16    CategoryView(kind: "Grand total", score: $grand_total)
17    if upper_section.all_selected() && lower_section.all_selected(){
18        // must call .init() so that Markdown string interpolation works
19        Button (.init("*Game over: \$(grand_total) points*\nClick to restart"),
20                action:restart)
21                .font(.title)
22                .buttonStyle(.bordered)
23    }
24    }.environmentObject(selectColor)
25 }

```

The internal states (sources of truth) for this application are kept in instances of two classes:

- `Dice` whose properties are
 - an array of 5 `Int`s with the current value of each die
 - an array of 5 `Bool`s indicating whether the die at the same index is fixed or not
 - function `roll` which sets a random number between 1 and 6 to all dice that are not fixed
 - function `clear_fixed` which indicates that no die is currently fixed

[\[source code\]](#)

```

1 class Dice:ObservableObject {
2     @Published var dice = [Int](repeating: 0, count: 5)
3     @Published var fixed = [Bool](repeating:false, count:5)
4
5     func roll(){...}
6     func clear_fixed(){...}
7 }

```

- `Section` whose properties are
 - an array of `String` for keeping the names of the categories of this section
 - an array of `Int` (having the same length as the names) for the current value of the score associated with a name; this value is initially -1 to show that no score has yet been computed
 - an array of `Bool` (same length as the names) indicated if this category has been selected
 - function `set_scores` for setting the values of the scores

- function `clear_unselected` for indicating that unselected scores should be reset to -1
- function `all_selected` for indicating that all its scores have been selected
- function `total` for computing the total of selected categories

[\[source code\]](#)

```

1  class Section:ObservableObject {
2      var names:[String]
3      @Published var scores:[Int] = []
4      @Published var selected:[Bool] = []
5      init(_ names:String){...}
6      func init_scores(){...}
7      func set_scores(values:[Int]){...}
8      func clear_unselected(){...}
9      func all_selected()->Bool {...}
10     func total() -> Int {...}
11 }

```

As properties of instances of these classes will be shared among many views, they cannot be annotated as `@State` with the corresponding `@Binding` which can only be used for *simple* values. So these classes inherit from the `ObservableObject` protocol. This implies that the programmer must ensure that the `objectWillChange` method is called on any modification of the object. But thanks to the `@Published` attribute, this happens automatically when references are made to this object. So the `Dice` class *publishes* its `dice` and `fixed` properties while the `Section` class *publishes* its `scores` and `selected` properties that need to be used externally. Note the `names` of the section are not needed outside the `Section`, so this list is not marked as an attribute, but as an *ordinary* local variable to the class instance.

Instances of these classes will be used in specialized views. Here is the `DiceView` which itself uses a view for a single die. [\[source code\]](#)

```

1  struct DieView:View {
2      @Binding var value:Int
3      @Binding var fixed:Bool
4      @EnvironmentObject var selectColor:SelectColor
5
6      var body: some View {
7          Image(systemName:value==0 ? "squareshape":("die.face."+String(value)))
8              .resizable()
9              .aspectRatio(contentMode: .fit)
10             .frame(width: 50, height: 50)
11             .background(fixed ? selectColor.color : .clear)
12             .onTapGesture {fixed.toggle()}
13         }
14 }

```



```

28         action()
29     }
30 }
31 }
32 }.padding(.vertical).border(.black)
33 }
34 }
35 }

```

A `CategoryView` displays its `kind` (a `String`) with the corresponding score (a dash if is negative). Its value is not modified inside this view, but it can be modified by the parent view (see function `update_totals`, lines 36-46 of next listing), so it is annotated as `@Binding`. `SectionView` gets a `Section` instance, an `ObservableObject`, as a parameter so its declaration must be annotated with `@ObservedObject`. Once the selection has been done, the scores must be updated in the main view by the caller. These actions are wrapped in a closure parameter³. The body of the `SectionView` creates a `CategoryView` for each name of its section. When it is tapped, the category is marked as selected and its background color is changed.

With these views, the application is built as follows [[source code](#)] [[source code of yahtzeeScores](#)].

```

1  struct ContentView: View {
2      @State private var move:String = "Roll"
3      @State private var nbRolls:Int = 0
4      @StateObject private var dice = Dice()
5
6      @StateObject private var upper_section
7          = Section(["Aces", "Twos", "Threes", "Fours", "Fives", "Sixes"])
8      @State private var upper_total:Int = 0
9      @State private var upper_bonus:Int = 0
10
11     @StateObject private var lower_section
12         = Section(["3 of a kind", "4 of kind", "Full house",
13                 "Small straight", "Large straight", "Yahtzee", "Chance"])
14     @State private var lower_total = 0
15     @State private var grand_total = 0
16
17     @StateObject var selectColor = SelectColor()
18
19     func roll_dice(){
20         nbRolls += 1
21         if nbRolls <= 3 {
22             dice.roll()
23             if nbRolls == 3 {
24                 show_scores()
25                 move = "Score"
26             }

```

```

27     }
28 }
29
30 func show_scores(){
31     let (upper,lower) = yathzeeScores(dice: dice.dice)
32     upper_section.set_scores(values:upper)
33     lower_section.set_scores(values:lower)
34 }
35
36 func update_totals(){
37     upper_section.clear_unselected()
38     lower_section.clear_unselected()
39     upper_total = upper_section.total()
40     upper_bonus = upper_total >= 63 ? 35 : 0
41     lower_total = lower_section.total()
42     grand_total = upper_total + upper_bonus + lower_total
43     dice.clear_fixed()
44     move = "Roll"
45     nbRolls=0
46 }
47
48 func restart(){
49     upper_section.init_scores()
50     lower_section.init_scores()
51     update_totals()
52 }
53
54 var body: some View {
55     ... see the code at start of section ...
56 }
57 }

```

The status of the system is kept in the following *simple* variables (of *value* type in Swift terminology) annotated as `@State` :

- `move`: label for the `Button`: either `"Roll"` or `"Choose\nscore"` (line 2)
- `nbRolls`: tracks the number of rolls (between 0 and 3) (line 3)
- `upper_total`, `upper_bonus`, `lower_total`, `grand_total`: tracks the various scores at this point of the game (lines 8,9,14,15)

There are three `@StateObject`s, ignoring `selectColor` for the moment:

- `dice`: an instance of `Dice` (line 4)
- `upper_section`, `lower_section`: instances of `Section` with different names. (lines 6,11)

The body of the `view` is vertical stacking of 8 views, the first one being a line with the dice and two buttons. The

action associated with the first button tracks the number of rolls and displays the scores at the third. The second button, used to change the color of the selection, will be presented later.

Then are added the upper and lower sections followed by two `CategoryViews` for their partial scores. When all categories are fixed, then a `Text` is added at the bottom.

The code for the `show_scores` and `update_totals` is straightforward is not discussed here as I want to focus on the SwiftUI interface aspects.

9.3. Accessing the global environment

The [Environment](#) is a mechanism used by SwiftUI to propagate values from a *View* to its descendants. It is used implicitly for handling some methods that are available on all view types. When these types of function are called their effect is to change a value that is available not only in the current view but also in all embedded views of this one. There are about 50 predefined [EnvironmentValues](#) dealing with global objects (e.g. `locale`, `timeZone`), characteristics of the display (e.g. `colorScheme`), text styles (e.g. `font`, `lineSpacing`), view attributes (e.g. `backgroundStyle`).

But it is also possible to define custom environment values to transfer information from one view to another without passing them as parameters as this can become cumbersome in complex applications.

In my simple application, this is probably an *overkill*, but I want to give an example of the use of an environment object. I define a color used for a *fixed* die or for a *selected* category (see the last part of the last figure). This color can be changed by clicking the `c` button. There are five choices of color that are selected in rotation. The environment is an instance of an `ObservableObject` with `@Published` properties and kept in the `ContentView` as a `@StateObject`. The object tracking the current color is follows [\[source code\]](#)

```
1 class SelectColor:ObservableObject {
2     @Published var color:Color = .yellow
3     let colorChoices:[Color] = [.yellow,.teal,.pink,.green,.orange]
4     var current = 0
5
6     func nextColor(){
7         current = (current+1)%colorChoices.count
8         color = colorChoices[current]
9     }
10 }
```

For setting the environment, an instance of this class is given as a parameter to the global view modifier `environmentObject` (see line 24 of [ContentView](#)). When the user taps on the `c` button, the `nextColor` method of this object is called to change the value of the `@Published` variable `color`. This *new* color is used in the current view for setting the color of the text for the end of game, but also in the `DieView` and `SectionView` which refer to this environment object with the following attribute.

```
1 @EnvironmentObject var selectColor:SelectColor
```

```
@EnvironmentObject var selectColor: UIColor
```

In the code of the views, the color itself is referred to `selectColor.color`. So when the value in the environment object is changed, this value is propagated to all views in the hierarchy. As for global variables, this is useful for making changes to the whole application, but it should be used parsimoniously as it implies recreating many views.

9.4. Changes for iOS 17

[[source code](#)] file names are the same as the previous ones, but suffixed by `_17`

Starting with iOS 17 and macOS 14, SwiftUI provides support for [Observation](#), a Swift-specific implementation of the observer design pattern implemented by means of macros that modify the abstract syntax tree of the program. The compiler can then track changes in objects and in the environment. This [migration guide](#) illustrates the changes on a simple example.

In our case, the following changes are made for iOS 17:

- Variable declarations annotated by `@StateObject/@ObservedObject` are annotated by `@State`
- `@EnvironmentObject` annotations are replaced by `@Environment(class_name.self)`
- The `@Binding` annotations stay the same
- Class inheriting from `ObservableObject` is now annotated with `@Observable`
- `@Published` annotations are removed
- The function call `.environmentObject(...)` is replaced by `.environment(...)`

The logic of the application is not changed, the main advantage is that any state is annotated the same independently of the fact that it is a single variable or an object. It is the job of the compiler to detect changes in the object and redraw only the necessary views

10. Conclusion

This document has presented small SwiftUI applications featuring some aspects of SwiftUI. After presenting how SwiftUI differs from the usual approach to developing interactive applications, some *unusual* features of Swift such as *trailing closure*, *property wrapper* and *result builder*, were first described as they are used extensively to define views in SwiftUI. State management and the associated bindings being at the core of SwiftUI, they were first illustrated with a single view application. Then more complex state management attributes were presented and put in action in an application using multiple objects and views.

This document does not deal with many aspects of SwiftUI such as scene management, animation or navigation lists, but I feel it gives the necessary tools to tackle them, because state management is at the core of SwiftUI.

References

REFERENCES

- Chris Eidhof and Florian Kugler, *Thinking in SwiftUI, a Transition Guide*, 1st edition, [2nd edition](#)
Very informative, but you should read the first edition (unfortunately not unavailable on the web...) before tackling the second.
- Mateus Rodrigues: Understanding SwiftUI : [View](#) ; [Modifiers](#)
Very clear and to the point explanation of views and modifiers
- Aleksandr Gaidukov: [How to Approach Wrappers for Swift Properties](#)
High level and clear explanation of property wrappers
- Mike Zaslavskiy, Anthony Williams, Ryan Zhang: [How the SwiftUI View Lifecycle and Identity work](#)
Very good explanation of the Swift state management works.
- Paul Hudson, [What's the difference between @ObservedObject, @State, and @EnvironmentObject?](#)
Short and to the point introduction to SwiftUI attributes with a good illustration of their differences.

Appendix: Useful tricks

APPENDIX. USEFUL TIPS

Here is a list of personal *non-obvious* techniques that I found useful during my SwiftUI development.

- To find all predefined controls, system and images in Xcode:
CMD-Shift-L : show Library or click the `+` in the upper right of the application window
- To see how the various views are organized add `.border(.red)` to the views
- From the first edition of [Thinking in Swift](#) (p. 10)
To inspect the underlying type of the body, use the following helper function:

```
1 extension View {
2     func debug() -> Self {
3         print(Mirror(reflecting: self).subjectType)
4         return self
5     }
6 }
```

The function is used like this to print out the view's type when the body gets executed:

```
var body: some View { VStack { */\*... \*/ } .debug() }
```

- within the body of a View it is possible to execute some code with the following pattern

```
1 { // some arbitrary code...
2     return {
3         View definition as usual
4     }
5 }
```

- From the second edition of [Thinking in Swift](#) (p. 45)

Insert `print` in a `view` body while ignoring the `void` return type...

```
let _ = print("Executing <MyView> body") // print when the body is re-executed
let _ = Self._printChanges() // print why the body is re-executed
```

- *Simulate* a callback function in a `view` by adding a function parameter such as

```
1 let action: ()->Void
```

that can be called from within the view to execute code in the caller context. The view is then created with `theView(action: the function)` or with a trailing closure.

1. My text editor claims that this document is less than a 20-minute read, but it surely does not consider making sense of the computer code... [↵](#)

2. I often wonder if some of these features were not introduced primarily for the sake of SwiftUI. [↵](#)

3. In principle, updating scores could have been done with a SwiftUI *simultaneous gesture*, but in SwiftUI the gesture of the caller is executed before the one of the callee, a dubious choice... This is not what is needed in my case because the selection must be considered for computing the totals before clearing them. I would have needed to *bubble* in JavaScript parlance, this explains the *callback trick*. [↵](#)