CHAPTER TEN : CREATE TIMER MODEL

I. INTRODUCTION

In the last chapter we designed the main storyboard of the PowerFocus app, which is the view in MVC (Model-View-Controller) architecture. In this chapter, we are going to code the model. The model in MVC is WHAT we want to represent in the view, and the modeling process is one of the most important steps in the whole development. A well designed model can make the app run really smoothly and robust. A poorly designed model can make it really hard to implement the controller.

Unlike designing the view, sometimes there is a fine boundary between model and controller. A bad decision in the model design can violate the MVC architecture. The most important question is: All UI dependent functions and data structures have to be in the controller, not the view.

Another simple rule is to ask you this question: Can I change my model to a completely different interface without changing a single line? If your answer is no, there might be some controller's functions inside your model. The bottom line is: whatever your GUI architecture is (MVC, MVP, MVVM…), your model should be completely UI independent.

A great tool to use in modeling is UML (unified modeling language). If you want to know more about UML, you can have more information in the following page:

https://www.omg.org/spec/UML/About-UML/

In this chapter, we are going to use the class diagram, you can use Star UML to create a class diagram. You can download Star UML in their website.

II. SPECIFICATION

The timer that we use is a countdown timer that switch between work and breaks (short break and long break). After a countdown goes to 0, the timer should switch to the following activity. There is a possibility to take long breaks after several work-break cycles, it can be enabled or disabled.

In order to create a model, you should analyze those previous requirements. From these requirements, you should come up with some specifications that describe the data structure and the actions. Here is a short list of better formed descriptions that are closer to data structure specification.

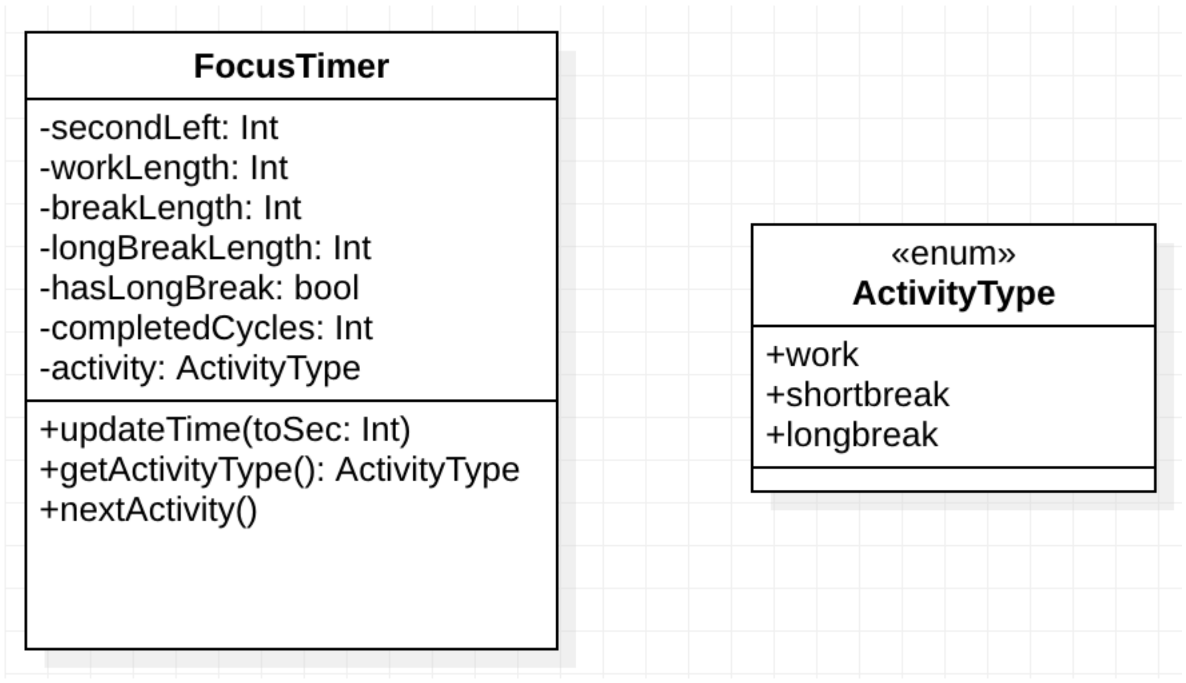
- Work time in minute
- Short break time in minute
- Long break time in minute
- Time elapsed in the current activity
- Whether the long break is enabled
- Frequency of the long break
- Number of completed work-break cycles
- Update the time of the current activity
- Go to the next activity

In this app the model is relatively simple and easy to implement. In other larger applications, the model can be huge and very complex. For these applications, designing and implementing the model will take most of the workload. If you have your own idea, you can also follow this approach for your own implementation.

You have to know OOP (Object Oriented Programming) really well and some design patterns that comes with OOP. The more experience you have in modeling, the better you model will be, and the easier your controller implementation will be.

III. TIMER CLASS DIAGRAM

According to the specifications in the previous section, we can define the following class diagram for the timer:

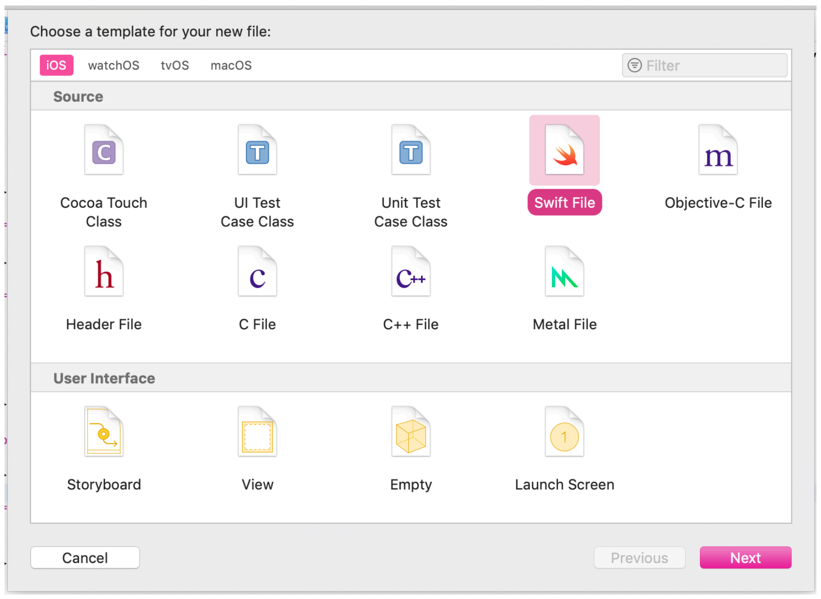


The ActivityType enumeration is an auxiliary structure for the FocusTimer class. It designed to represent different states of activities: WORK, BREAK, and LONGBREAK.

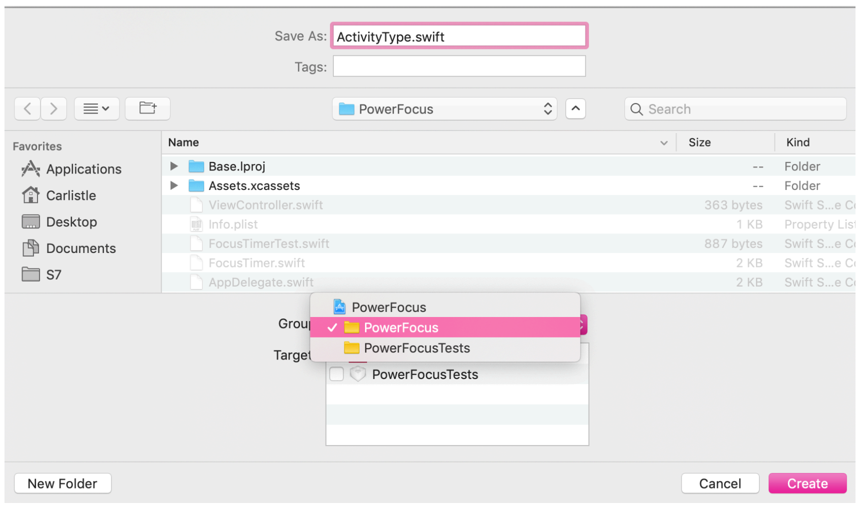
In the main model class FocusTimer, we need to memorize and determine the current state of activity. In FocusTimer class, the controller calls the method updateTime with the time in second. The method getActivityType returns an ActivityType to allow the controller to update the theme according to the ActivityType. The method nextActivity is a private method that reinitialize the seconds and change to the next activity.

IV. ENUMERATION ACTIVITYTYPE

In this section, we are going to review enumeration that is discussed in chapter 7 section 3. If you have forgot what an enumeration is, it's highly suggested to go back and review the syntax. To create a new enum file, in the menu select **File > New > File** or use the shortcut **Command(⌘) + N**. Under iOS, choose **Swift File**, then click next.



Enter ActivityType.swift as the file name. Then save this file in the group PowerFocus, not in the root group.



Save As: ActivityType.swift

Tags:

Name	Size	Kind
Base.lproj	--	Folder
Assets.xcassets	--	Folder
ViewController.swift	363 bytes	Swift S...e C
Info.plist	1 KB	Property List
FocusTimerTest.swift	887 bytes	Swift S...e C
FocusTimer.swift	2 KB	Swift S...e C
AppDelegate.swift	2 KB	Swift S...e C

Group: ✓ PowerFocus

Targets: PowerFocusTests

The syntax of an enum declaration is:

```
enum name {
    case case
}
```

where the name of the enum is name and the cases are declared in the block. An enumeration can adopt a protocol. The names of the adopted protocols are put after name, separated by commas. To make the enumeration ActivityType easier to access and easier to compare, it adopts the String protocol. All primitive types (integer, double, string, char, bool) have a protocol for other classes or enumerations to adopt.

```
enum ActivityType: String
```

There are three cases: work, shortBreak, and longBreak. Since this protocol adopts the String protocol, we have to assign a string value to each case. For instance, for the case work:

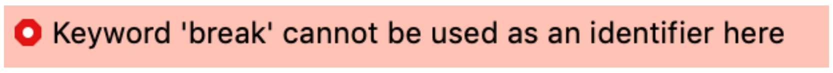
```
case work = “work"
```

To access a case work, we can use ActivityType.work or just .work in some cases thanks to Swift’s strong type inference.

I recommend you to try to write it yourself, you can compare your solution with the following declaration:

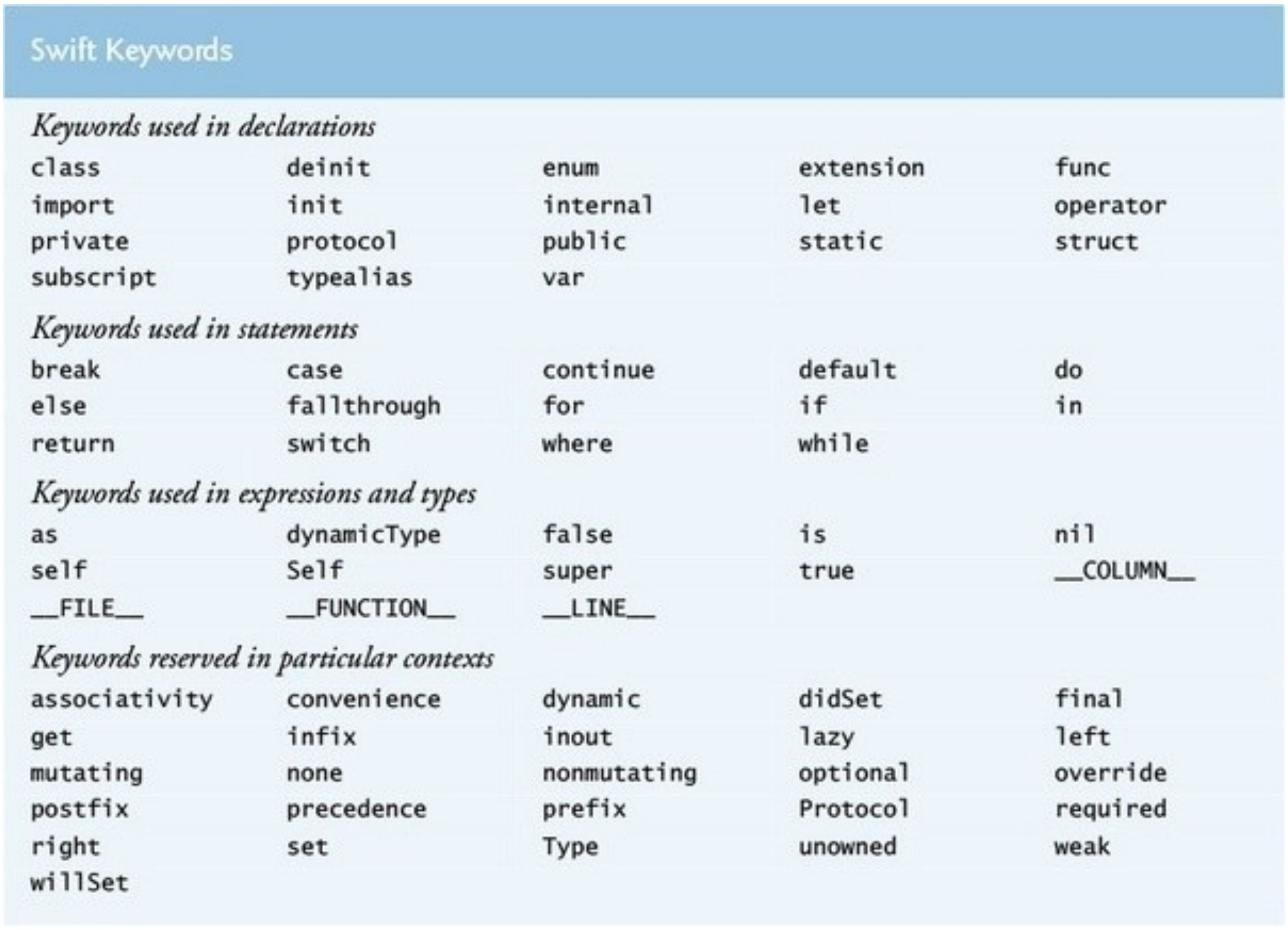
```
import Foundation

enum ActivityType: String {
    case work = "work"
    case shortBreak = "shortBreak"
    case longBreak = "longBreak"
}
```

One thing to notice is that we name the case shortBreak instead of break. This is because some keywords are reserved for the Swift compiler. A case can’t be declared as break because break is a keyword used in statements. You will see the following error message if you try to declare a case called break:



Keyword 'break' cannot be used as an identifier here

Here are some Swift keywords, you don't have to memorize these by heart, but you have to know how to avoid using them in your declaration.



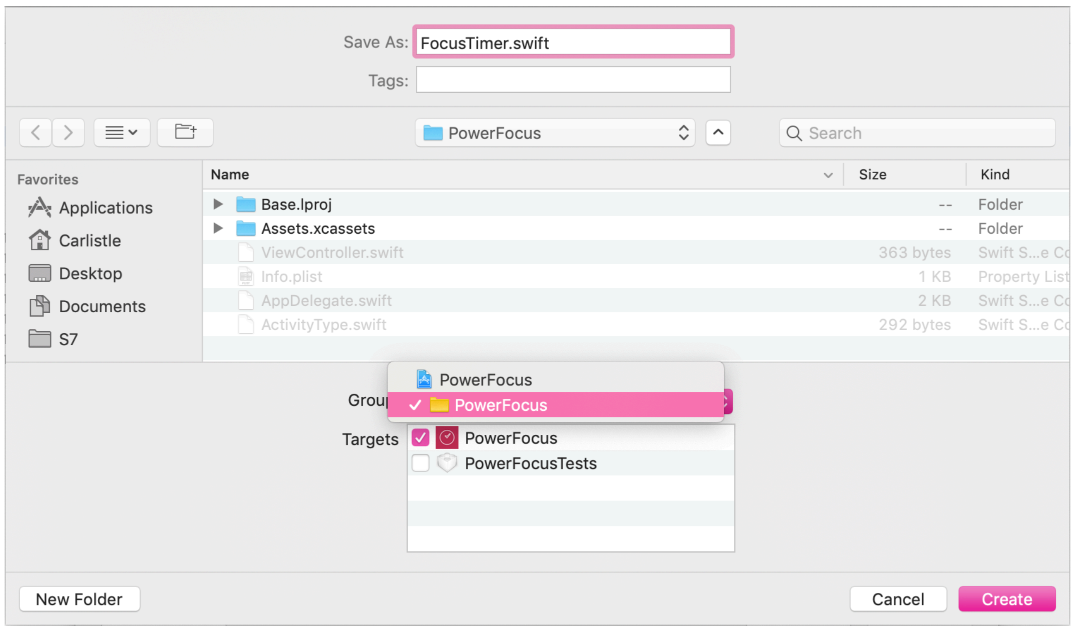
Swift Keywords

Keywords used in declarations				
class	deinit	enum	extension	func
import	init	internal	let	operator
private	protocol	public	static	struct
subscript	typealias	var		
Keywords used in statements				
break	case	continue	default	do
else	fallthrough	for	if	in
return	switch	where	while	
Keywords used in expressions and types				
as	dynamicType	false	is	nil
self	Self	super	true	__COLUMN__
__FILE__	__FUNCTION__	__LINE__		
Keywords reserved in particular contexts				
associativity	convenience	dynamic	didSet	final
get	infix	inout	lazy	left
mutating	none	nonmutating	optional	override
postfix	precedence	prefix	Protocol	required
right	set	Type	unowned	weak
willSet				

V. CREATE CLASS FOCUSTIMER AND ATTRIBUTES

Just like how we created an enum, create a new file in the menu select **File > New > File** or use the shortcut **Command(⌘) + N**. Choose Swift file and click next.

Name the file FocusTimer.swift and put it in the group PowerFocus. For this class, we need to create some tests. But for now, do not check the target PowerFocusTests for now. We are going to put the test class in this target.


Save As: FocusTimer.swift

Tags:

PowerFocus

Name	Size	Kind
Base.lproj	--	Folder
Assets.xcassets	--	Folder
ViewController.swift	363 bytes	Swift S...e Co
Info.plist	1 KB	Property List
AppDelegate.swift	2 KB	Swift S...e Co
ActivityType.swift	292 bytes	Swift S...e Co

Group: ✓ PowerFocus

Targets: [✓] PowerFocus [ ] PowerFocusTests

The syntax of class declaration is:

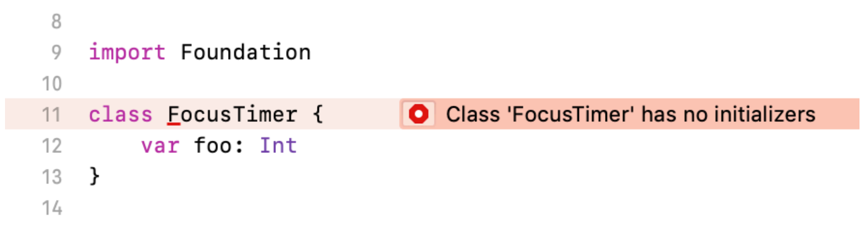
```
class <#name#>: <#super class#> {
    <#code#>
}
```

The class FocusTimer doesn't have any super class, so the signature is: class FocusTimer{...} For the variables of a class, the visibility options are open, public, private, fileprivate, and internal.

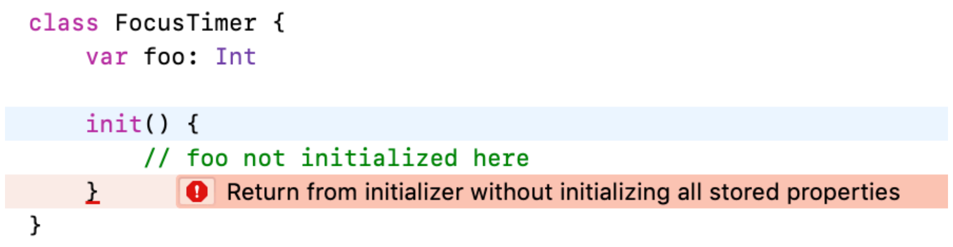
All entities in your code have a default access level of internal if you don't specify an explicit access level yourself. As a result, in many cases you don't need to specify an explicit access level in your code.

We are going to use a special scope called **private(set)**. This type has a public get access and a private set access. In the most of the object oriented languages like C++ or Java, the equivalent of a private(set) variable in Swift is a private variable with a public getter.

It is also possible to provide an attribute with a default value. If there is a variable that doesn't have a default value, and it is not initialized in all constructors, there will be an error like this:



```

import Foundation

class FocusTimer {        Class 'FocusTimer' has no initializers
    var foo: Int
}

```



```
class FocusTimer {
    var foo: Int

    init() {
        // foo not initialized here
    }        Return from initializer without initializing all stored properties
}
```

Although a class doesn't have to have any init constructor, it is a good practice to declare one or more init constructor(s). In init, we can have control of how an object of the class is set up. If a variable foo has a default value, it will be set to its default value after the creation even though it is not specified in the constructor

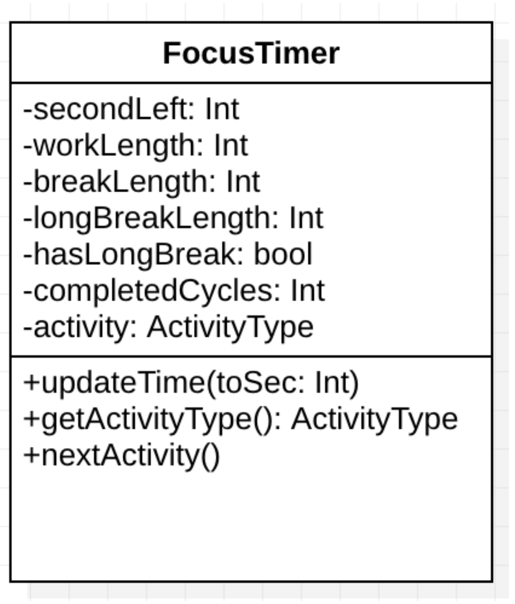
```
class FocusTimer {
    var foo: Int = 1

    init() {
        // foo not initialized here
    }
}
```

This class can compile and gives the following result:

```
    let foo = FocusTimer()   // create an instance using init()
    print(foo.foo)           // will get 1
```

Take another look of the FocusTimer class diagram, in the attribute section, we can define the attributes like this:

```
    // Length of activities
    private(set) var workLength: Int
    private(set) var breakLength: Int
    private(set) var longBreakLength: Int

    // number of completed work-(long)break cycles
    private var completedCircles = 0

    // number of cycles between each long break
    private(set) var longBreakFreq: Int

    // If the timer enables long break cycles
    private(set) var hasLongBreak: Bool

    // The current activity type
    private var activity: ActivityType
    // current activity's time left
    private var secondLeft = 0
```



The variable secondLeft is an internal counter of how many seconds are there left in the current activity, its default value is 0 second. It is used and updated in the method updateTime(to sec: int)

Long break frequency and enable flag should be private set because only the timer itself can set these attributes but other classes are allowed to get this information.

The completedCycles counts the number of completed work-(long) break cycles. This is useful to determine if the next break is a long break and to update the cycle count. It definitely should be private since we don't want other classes to get or set this counter.

The lengths of the activities are integers with private set visibility. The controller should be able to get it to set the countdown before an activity begins.

V. FOCUSTIMER'S INITIALIZER

Constructors get called every time a new instance of the class gets created. They control the new instance's state and initialize all the sub-objects. All non-optional attributes without a default value should be initialized in a constructor. Optional attributes without a default value will be nil if they are not initialized in a constructor. Writing a constructor is really important in OOP and sometimes really hard. We should only initialize the object in a constructor, because everything else should be encapsulated in methods.

This FocusTimer class is relatively easy and straightforward to initialize. Here is its constructor:

```
init(_ workLg: Int, _ breakLg: Int, _ longBreakLg: Int, _ longBreakFreq: Int = 3,
     _ hasLongBreak: Bool = true) {
    self.workLength = workLg
    self.breakLength = breakLg
    self.longBreakLength = longBreakLg
    self.longBreakFreq = longBreakFreq
    self.hasLongBreak = hasLongBreak
    self.completedCircles = 0
    activity = .work
}
```

In the parameter list of init, we can use "_" for the external name of a parameter to indicate that there's no external name. The caller should use the same name for both the external and internal parameters.

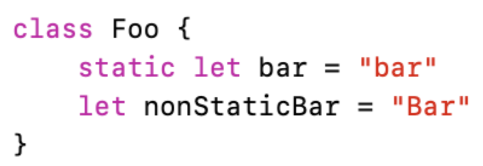
Like a class attribute, parameters can also have a default values by putting "= default value" after the parameter type. This is convenient for the user because they only have to specify the mandatory parameters in the parameter list. If parameters with default values aren't specified, it will take the default value instead of nil.

The order of mandatory parameters is not mutable, but you can specify the parameters with optional values in any order you want. But you can't put a parameter with a default value before a mandatory attribute because the compiler uses different policies calling different types of parameters (by order and by name)

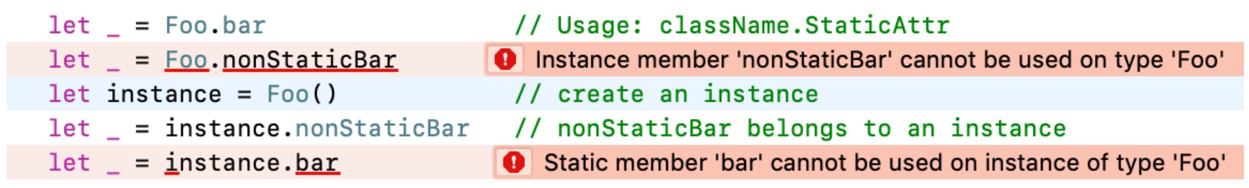
VI. STATIC ATTRIBUTES

In the body of this constructor, we initialize the attributes to the values of the parameters. A great practice in OOP (and other programming in general) is to avoid hard-coded values. Hard-coded values can be hard to find and synchronize. They can cause a lot of troubles for testing and maintenance. In this constructor, longBreakFreq's default value, hasLongBreak's default value, and self.activity are hard coded values. The initial value of completedCircles is 0, which is a hard-coded value. But it is inevitable because when an instance is created, it's number of completed cycle is 0.

A common solution to this problem is to use static attributes. A static attribute doesn't belong to an instance like normal attributes do, it belongs to the class. Here is an example of a class Foo has a static attribute bar:



```
class Foo {
    static let bar = "bar"
    let nonStaticBar = "Bar"
}
```



```
let _ = Foo.bar                  // Usage: className.StaticAttr
let _ = Foo.nonStaticBar         ❗ Instance member 'nonStaticBar' cannot be used on type 'Foo'
let instance = Foo()             // create an instance
let _ = instance.nonStaticBar    // nonStaticBar belongs to an instance
let _ = instance.bar             ❗ Static member 'bar' cannot be used on instance of type 'Foo'
```

The static variable bar can only be accessed by calling the class: **Foo**.bar. There's only one string in the memory for the class. However, for each instance of class Foo, it has its own version of the variable nonStaticBar. As a result, a non-static attributes can't be accessed using class because there are many possible versions that belong to different instances of the class Foo.

The default value of some attributes should be the same among all instances of class FocusTimer. If an instance changes a default value, the default values of other instances should also be changed. In this case, these default values no long belong to an instance and they become static values of the class. Otherwise, changing the default values can be really hard, here is an example of equivalence in Java since Swift doesn't allow us to do such modification:

```
instance.getClass().getAllInstances().forEach(
        obj -> obj.nonStaticBar = newValue
 );
```

Another important reason why we make it static is the memory efficiency. Memory efficiency is a very important criteria of code quality, and it is very important in mobile app development.

For a certain class, unless defined otherwise in the constructor, you can instantiate as many instances as you want. Each object has its own segment in the memory heap. The memory heap is a relatively limited resource comparing to the file storage size. Even for the latest models of iPhone and iPad, there is far less memory than a desktop computer like a MacBook or an iMac.

Say our class Foo has a million instances and each instance is 1kB. Then the total memory for these instances will be 1KB * 1,000,000 = 1GB, which is a quarter of the total memory of an iPhone 11 Pro. The code can be very inefficient in terms of memory.

If the class Foo uses static values, its one million instances don't have to allocate one million copies of the static values in the memory. There is only one copy of bar, which belongs to the class. The size of each instance can be dramatically reduced using static variables.

To define an attribute as static, use the keyword **static** in the signature. Now go back to the FocusTimer class and add these following static variables:

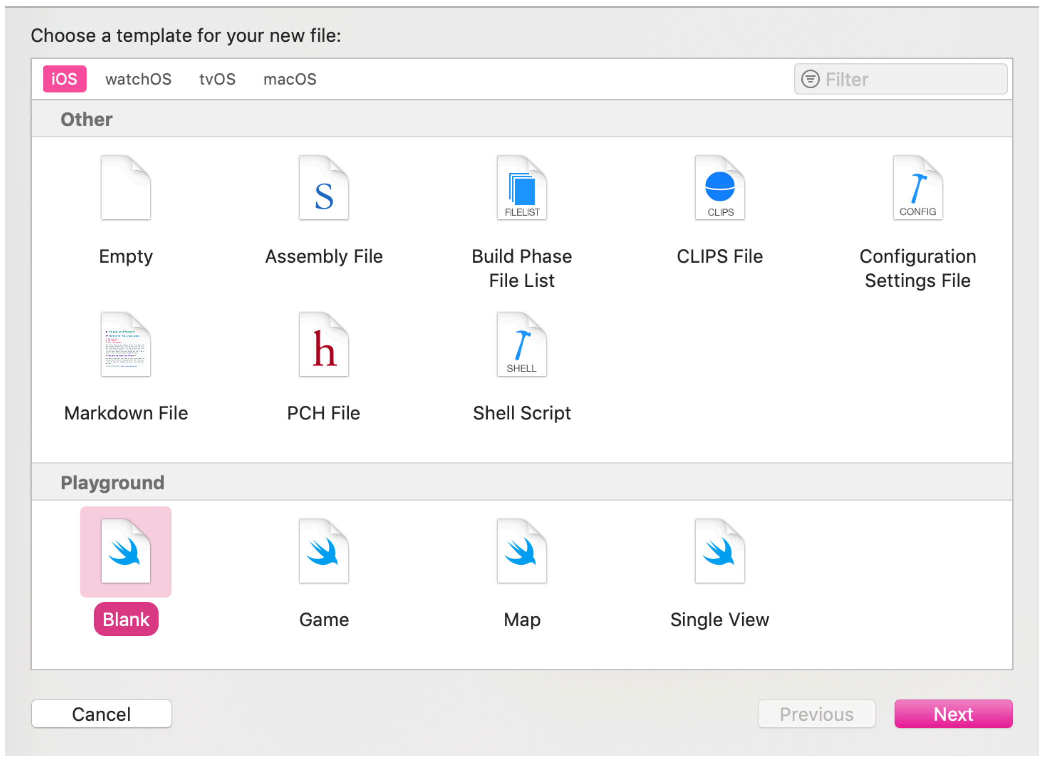
```
    // Static constants for constructor
    private static let defaultLongBreakFreq = 3
    private static let defaultEnableLongBreak = true
    private static let defaultActivity = ActivityType.work
```

The constructor becomes:

```
    init(_ workLg: Int, _ breakLg: Int, _ longBreakLg: Int, _ longBreakFreq: Int =
FocusTimer.defaultLongBreakFreq,
         _ hasLongBreak: Bool = FocusTimer.defaultEnableLongBreak) {
        self.workLength = workLg
        self.breakLength = breakLg
        self.longBreakLength = longBreakLg
        self.longBreakFreq = longBreakFreq
        self.hasLongBreak = hasLongBreak
        self.completedCircles = 0
        activity = FocusTimer.defaultActivity
    }
```

VI. EXERCISE

You can do these exercises outside the PowerFocus project by using Xcode playgrounds. In File > New > File, under iOS, at the end of the list there are playground options. These two following exercises are considered as models, so you will need a **Blank Playground**.



EXERCISE #1

We have seen a cool feature of Swift: **private(set)** that provides public get access but private set access. This means that we can get the value of such variable outside the class, but we can't set its value.

An analogy of this type of variable is an item in a shop's window. You can admire it(get method) freely from the outside, but you can't play with it(set method) from the outside. But once you are inside the shop(in the class), you have full access of everything.

In this exercise, we are going to implement in a class **Foo,** a private(set) variable **bar** without using a private(set) declaration. There are two simple ways to implement this:

The first approach is to use a private variable bar. The convention is to get and set a private variable t of type T is:

```
func getT() -> T {
    return self.t
}

func setT(to newValue: T) -> Void {
    self.t = newValue
}
```

The scope of access depends on the access of these two methods. If these two methods are public, the private variable T "becomes" a public variable. Use a public getter for a private variable makes it equivalent to a private set variable.

The second approach is to use computed values, which do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly. A syntax example is:

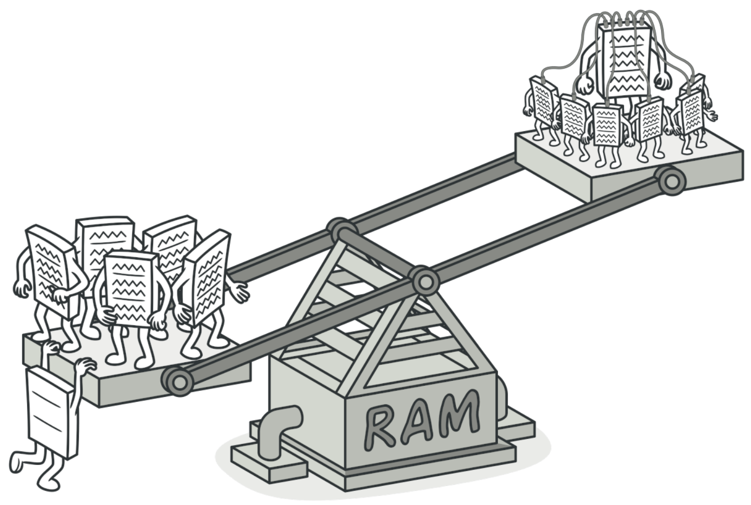
```
// the variable bar an Int but it is not stored but
// computed each time it is used
// If it's used as a rValue, get is used
```

```
    // If it’s used as a lValue, set is used
    var bar: Int {
        get {
            // returns some int value
        } set (newValue) {
            // set the value of bar to the newValue
             // but this assignment can be customized here
        }
    }
```

In a class FooTwo, define a computed value bar. To make it public set, define a get but not get. To store the real value, use another private variable realBar. RealBar should be used in the computed value.

EXERCISE #2

Using some static attributes can save much memory when a class is frequently instantiated. Another way to save memory in this type of situation is **flyweight**.



A flyweight is an object that minimizes memory usage by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the objects temporarily when they are used.

In this exercise, we are going to use flyweight for a class Glyph that encapsulates a character. It can be initialized from a character and it can be displayed by a display method.

The design pattern flyweight is used in a class GlyphFactory. This is a factory class for Glyph instances. A GlyphFactory should have a static instance of type GlyphFactory. A factory should only have private constructors and these constructors should only be used to initialize public static instances. To use a GlyphFactory, you should use GlyphFactory.INSTANCE.xxx instead of new GlyphFactory().xxx

To memorize and fabricate Glyph objects, there's a dictionary glyphCache[Character: Glyph] that stores Glyph instances. The constructor of the class GlyphFactory initializes the glyphCache map to empty.

The method getGlyph returns a Glyph object from a character. If this method is called with an unseen character (not in the keys of the cache), a new Glyph object is created then stored in the cache (and then returned as result). Otherwise, if we are asking for a char that is already in cache, we should return the value from cache and no new instance should be created.

Example: the sequence (1)a - (2)b - (3)a - (4)c - (5)c

In (1), (2), and (4), new instances are created.

In (3), and (5), get value from cache.

SUMMARY

In this chapter, we have worked on these topics:

- have an introduction to UML
- set up app specification
- create class diagram for the model
- declare enumeration
- declare class attribute
- declare class initializer
- use static attributes

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

If you can't figure out the exercises on your own, you can check the answers at

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Ten_Exercises.zip

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Ten_PowerFocus.zip