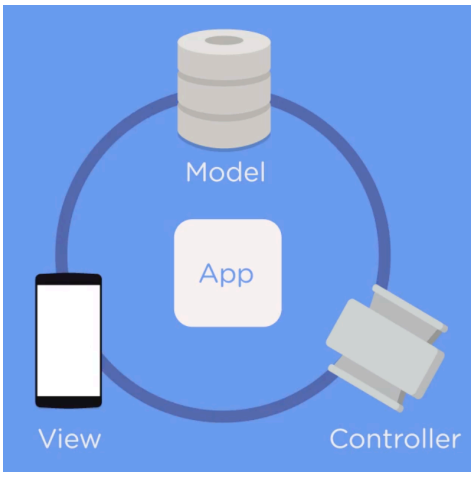
team treehouse: building a simple iPhone app with Swift 2.0

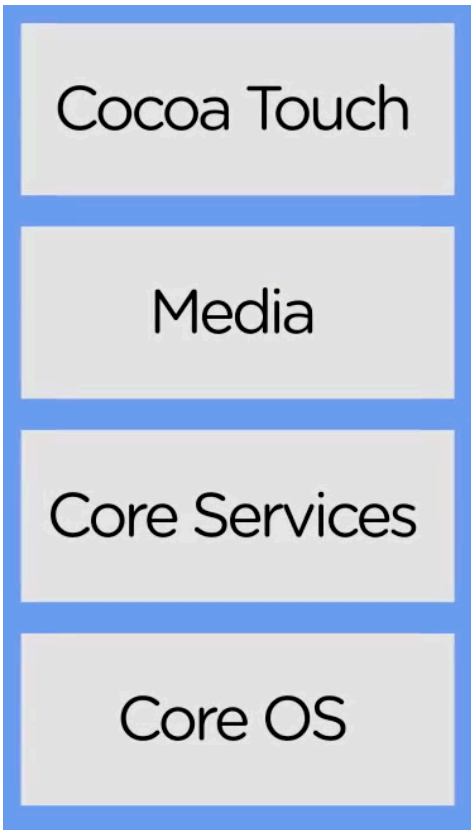
- there's an *established pattern* for *version numbers*
  * version numbers usually take on the form of
  * **Major.Minor.Patch**
- every app at it's basic level consists of **three main components**
  * models
  * views
  * controllers



- **the models** *contain the data* that we will use in the app
- **views** are *parts of the app* that we *see on the screen* and *interact with*
- **controllers** are sort of these *puppet masters* that *coordinate between the models* and *the views*
- **iOS operating system** *manages the device hardware* and *provides certain technologies* that we can use
- the **iOS SDK** or *software development kit* contains *all the tools* and *interfaces* that we need, to use the operating system to *develop*, *install* and *run our own apps*

- at the very *high level*, *iOS* sort of looks like this, we have
  * Cocoa Touch
  * Media

    * Core Services
    * Core OS



- these are all *different layers of code* and *each layer* builds *on top of the next one*
    * making it much easier to write code to achieve certain tasks
- you will often hear of iOS development referred to as **Coco Touch development** and this refers to the fact that we *interact a lot* with the *Coco Touch layer of code* when building our own apps
- when you work with the iOS code base, you will encounter many of its classes that Apple engineers themselves have written for us to use
- when writing Swift code, we use the *string type* to *represent a String*
- *Strings in iOS* are an *instance of* the *NSString class*
- how is a string a class?
    * swift is a brand new language so *none of this underlying iOS code* that we're going to use from Apple is written in Swift
        ** it is written in *Objective-C*
- *Objective-C* works a bit differently and *names of classes* are *prefixed with letters* to *identify the part of the operating system* they belong to
- for example : **NSString**

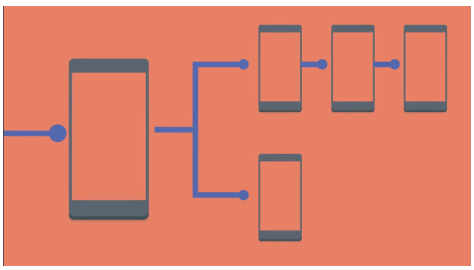
* the *prefix here* is *NS*
* similarly you will find
    ** NSNumber
    ** NSArray
    ** NSDictionary
    ** and so on
* now this *prefix is used because* this code is *part of a library of code* called **Foundation**
    ** which was written as part of the *Next Step operating system*
    ** the *prefix NS* refers to *Next Step*
    ** what is *Next Step*?
        ** back in 1985, after some turmoil in the company, Steve Jobs was *fired from Apple* and he ended up starting another company called *NeXT*
        ** *The NeXTSTEP operating system* was developed for one of their main products
            ** the *NeXT computer*
        ** much later, in 1996, Apple wasn't finding much success in their product line and the *decided to buy NeXT*
            ** this brought Steve Jobs into the company, and the *Next Step operating system* became the *foundation for almost all the Apple software that we use today*
        ** we are standing on a *lot of legacy code*
            ** this is code that has been around for a very long time
        ** so many of the things like *drawing text on screen* or *making buttons that tap*
            ** all of that stuff, we don't have to worry about because Apple as written all of the code to do all that stuff

xcode project

- **interface builder** is used to build *visual components of our apps* here
- **ViewController** file is where we will write *some logic to control how the app works*
- **AppDelegate** file contains *code that communicates with the operating system* and is *responsible* for *opening*, *closing* and *saving* the **state of the app** amongst many other things

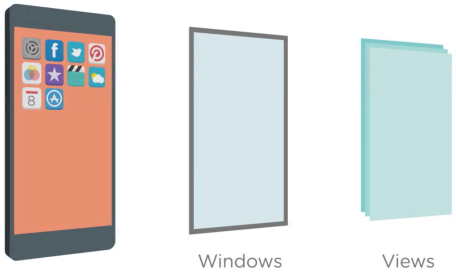
designing with interface builder

- when building the app, we have *three main components to work on*
  * *data*, or *the models* in the app
  * the *visual representation*, or *user interface* of the app
  * *controllers*
- **interface builder** does *two main things*
  * it allows us to *layout elements in our views* and *specify their position on the screen* among different devices
  * it involves interacting with these visual elements
  * when you use an app, you scroll the screen, tap buttons, select text, and do many other things
  * when you take actions like these, you *expect certain things to happen*
  * *interface builder* lets us *connect our visual elements* to *code* so we can *execute some action* when the *user interacts with the element*
- within *interface builder* we work in a *storyboard*
  * a **storyboard** is a *visual representation* of the *user interface* of an *iOS application*
  * *showing screens of content*, and *connections between those screens*



introduction to views

- a *view* represents a *an element*
- in *iOS*, you use *Windows* and *Views* to *present content* on the screen



- each app has *at least one window* and *acts as a container for many different views* that make up your app
- a **view** *manages a rectangular area* within this window in your app

View

- **views** are *not only responsible* for *drawing content on the screen but also* for *handling user interaction* like when we *touch our screen* and *managing the layout* of *any subviews within a particular view*
- **views** are *instances* of the **UIView** *class*
  * one of the classes in the code that Apple provides
  * you can use these views as *building blocks* when building the app
- *UIView* even has *specialised views* that let you put text, images and other types of content on the screen
- think of views as *building blocks* where you can use *multiple views* in *different layers* and *hierarchies* to build how the app will look like on the screen
- a *single view* can *contain any number of subviews* and we *arrange them in different ways* to create a **user interface**

frameworks and UIKit

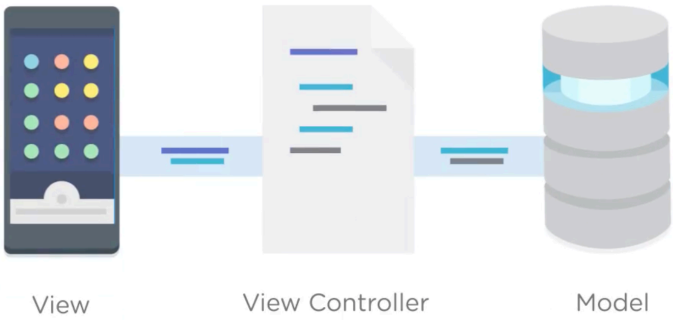
- a **software framework** is a *reusable code base* that *provides particular functionality* in the *context of a larger software platform*
- since Apple has written lots of code for lots of *different uses*
    * these are neatly *split up* into *several different frameworks*
- **UIKit** - the framework that deals with the *window and view architecture* to *manage* an app's *user interface*, *handles events* and *provides a means* for the app to *interact with the system*
- other frameworks include *Foundation* which *provides a lot of base objective C classes*
    * in Swift *base types* include String, array, dictionary and so on
    * the Objective C equivalents are NSString, NSArray and NSDictionary
        ** these classes are provided in the foundation framework
- there are lots more including frameworks like **Webkit** to *display web content* in *browsers*
- **Core location** for *location services* and *tracking*
- **Game kit** creates *social games* and many more
    * all of this code isn't bundled into the app by default
        ** we can *pick and choose* what Apple code we *work with* by *importing frameworks*
- there are *two ways* we can use a *label*, *button* or *many other components in UI Kit*
    * since they are *classes*
    * we can *create them in code* by *creating an instance of the class*
        ** each of these classes have *special initialisers* that *get them ready for use*
    * second way is through **interface builder**
    * for example *modifying* some of the *properties* in the *attributes inspector*
        ** Xcode automatically does the same thing in the background
            ** it *creates an instance of the correct class* and get to use that

introduction to view controllers

- *controller mediates between the view and the data*

view controller

- **view controller** is an *important component* of our app that *links the app's data* to its *visual appearance*



- most apps have a *fair bit of information* to *display in a limited amount of space*
  * these *showing* and *hiding* of *information* is *handled* by a **view controller**
  * and by having *different view controllers control different sections* of our *app's information*
    ** we can *seperate our code* into more *manageable chunks* and keep it somewhat *neat* and *tidy*
- a *view* is *controlled* by a *ViewController*
- typically *each main view* has a *backing ViewController* to *control* it

creating IBOutlets

```
@IBOutlet weak var funFactLabel: UILabel!
```

- we're creating a *variable stored property* with the name *funFactLabel* and it has *type UILabel*
- the first keyword there **@IBOutlet** is a *type qualifier*
  * this is a *tag applied* to a *property declaration* that *allows Interface Builder* to *recognise this property* as an *outlet* and it *synchronises* the *visual element* we have in our scene with the *stored property* that we *added in our code*
  * *IB* in the *IBOutlet* stands for *Interface Builder*
- the second keyword there **weak** has to do with *memory management*
  * *memory management* is a *complex topic* and a *very important one*

- *label* created in *interface builder* has a *weak relationship* with the *view controller* and that's why we have the *weak keyword*
    * *all IB outlets have weak relationships*
- the **exclamation mark** in the end of *UILabel*
    * when the *system loads views*
        ** not everything loads immediately at the same time
        ** things are loaded as they're needed to **optimise performance** and **memory** on the iPhone
        ** the *exlamation mark there* indicates that *there's a chance* that *our outlet won't be loaded by the time the view loads* and *if we try to access the property before that* **it will crash**
        ** *this exclamation mark* is a *Swift language feature* called **optionals**
            ** **optonals** are *really important* because they **allow us to set values to nil**
            ** **nil** represents **nothingness** and an *optional value* indicates to the compiler that *there's a chance* this value we're looking for *might not exist*
        ** the *optional* is used here for the *same reason*
            ** it *tells the compiler* that the *connection* between the *label* in *Interface Builder* and the *property in code might not exist yet because* it *hasn't been established* and that *it will happen* **once the view has been loaded**

using IBAction to execute methods

```
@IBAction func showFunFact() {

}
```

- this is a **common design pattern** when writing iOS apps known as **target action**

Target Action

- the *action is the message* that *control sends* and the *receiving object* is the *target*
- so in our case, the *view controller class* is the **target**, and *showFunFact* is the **action**
- **remember** that **all stored properties are scoped to the instance of the class and are available in the definition throughout**

asking for help

- a *common misconception* is that *developers know anything and everything* about *the code they're writing*
    * in reality *good developers* **know how to find answers quick as they come up against obstacles**
- what one *should be doing* from *the very beginning* is **developing the habit of looking at documentation**
- *anytime you stumble* and *can't figure out what to do*
    * **start with documentation**
        ** Apple does an *excellent job* and has a *dedicated team* to *cover every little aspect of the SDK* in their *documentation*

creating a data collection

- **data collection** are *data types* that *allow* you to *store* and *manage groups of values* or *objects*
- we learned about two collection types
  * arrays
  * dictionaries

views

- when the **view loads onto the devices screen**
  * it **notifies the view controller** that **it has loaded**
  * the **view controller** then **immediately runs the code in the viewDidLoad method**

refactoring into a model

- our app implementation works so far but *our code isn't structured well*
- even though we have very little code in our project
  * *it's hard to say what code sets up the visual appearance, what controls our views* and *what serves as our data model*
  * **it's important to keep these things separate** because as *the project grows* and *as you write more code* **having a same structure really helps write bug-free code**
- we need to **refactor our code**
  * what is meant by *refactor*?
    ** **refactoring is a process of restructuring code without changing any of its behaviour**
      ** so our app will work *exactly the same* after *refactoring process* but **our code will be organised better**
- *in an app* **the less code you can put into a single object and a single file, the better**
  * one of the *main reasons we want to do this*, to **seperate the data** that's in our *view controller*, is *to make both our class and model a lot more flexible*
  * *building a strong relationship between two objects* while it might seem like the best tragedy, comes with *plenty of pitfalls*

** those objects *can only be used with one another* and *in general*, **we want things to be more flexible and reusable**

more on frameworks

- **a framework can thought of as reusable code for common tasks that we can then build on top of to create our applications**
  * we mentioned earlier that a lot of the code that Apple has built over the years is broken app into different frameworks that we can add to our projects when we need

```
1 import Foundation
2
```

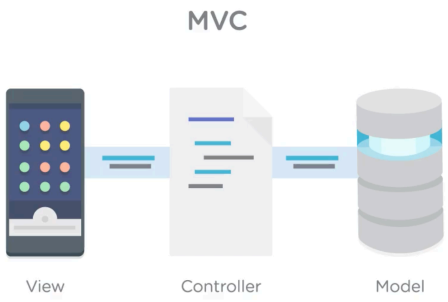
- this *import statement* at the top *lets us use code from that specific framework* **within the swift file that we imported in**
  * so by importing *Foundation* there
    ** you can use any of the classes *from the Foundation framework*
    ** but since **the import statement is only in this file, Foundation is only available to the code in this file**
      ** *if I wanted to access it in a different file*
        ** *I can import it again elsewhere*

structs or classes

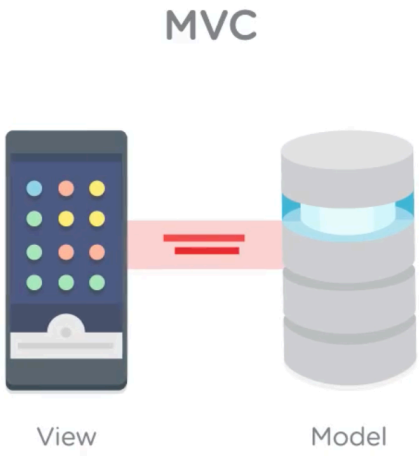
- we want to create an object that *encapsulates our facts in a simple model*
  * what do we use?
    ** a struct or a class?
- we said that we *should limit our use of structs for simple data* where we *don't need to worry about which particular instance we're working with*

model view controller

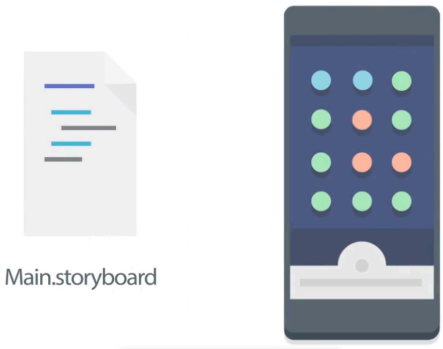
- software engineers often run into problems when writing code
  * *some of these problems are common enough* that there are *widely accepted solutions for them*
  * *these solutions* aren't bits of code that we write
    ** rather they're **ways of organising our code to make the components more flexible and reusable**
      ** these solutions are called **design patterns** and one of the *more fundamental design patterns in iOS development* is known as **MVC** or **the model view controller design pattern**
        ** let's take a look at each component in this pattern, let's start with **the model**
- all the data in our app as well as the logic involved in manipulating or transforming that data, goes into *either a Struct, Class or Enum*
  * we then *refer to these classes* as *models* or *model objects*
  * *the data* that *is contained in this object can come from anywhere*
  * where the data comes from doesn't matter as much because the object we create *serves as the data source for the rest of the app*
  * the *objet we create organises the data, performs some logic on it if it needs to* and *allows the rest of the app to access it*
  * *it's important to know* that this doesn't mean we have *just one single model in our apps*
    ** for our app, yes we will ave just the single model but *larger apps can have many more*
  * in a *hypothetical social networking app*
    ** a user can be represented by *one model object*
    ** stuff that they share like *photo albums* can be yet *another object* and so on
  * when we **follow the MVC pattern**



- the *model doesn't communicate with the view in any way*

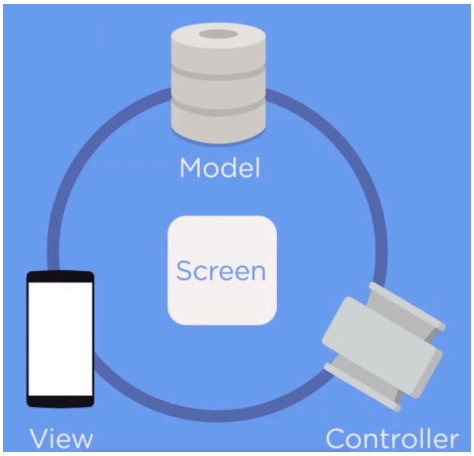


- getting the information *from the model* and *displaying it in the view* is the *controller's job*
  * by *separating the code like this, the model object can be re-used across other views* if we need to *without having to repeat the code* or *reimplement any functionality*
- after the *models*, we have **views**
  * the **views** in the *MVC pattern consist of* **everything that a user can see**
    ** stuff that builds up the *user interface of an app*
  * a **view's purpose** is to **draw on the screen** and to **display data from the model object** so that the **user can see and interact with it**
    ** in our application, *the views* are *represented* in the *Main.storyboard file* and *consists of the main view* along with the *labes and buttons* that we added

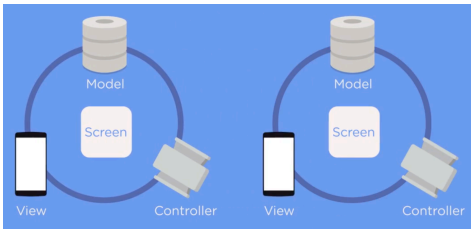


- just like models, we don't only have one view

    * we have *several views across the app*
- the *last part of this pattern* are the **controllers**
    * since the **views** are supposed to *visually represent data in the model* but **can't communicate with the model**
    * and **the model is supposed to handle manipulation of the data for the view but can't actually communicate with the view directly**, we *have this third object*, **a controller that mediates between the two**
- **the controller typically contains code to connect parts of the view to the model** and **to facilitate that communication between the two of them**
    * in our app, *view controller class is the controller*
        ** it *maintains a reference* to *both the view as well as the label and buttons we added throughout IB outlets*
        ** similarly *the data model is part of the controller* as well *through the stored property that we added*
        ** **inside methods** like **viewDidLoad**, the **controller pulls data out of the model** and **assigns it to properties on the view**, thereby **serveing its intermediary purpose**
- earlier when we *moved data out of the controller and into the model* we *were explicitly creating the model class to follow the MVC pattern*
- finally with the controller, just like the model, **this pattern does not restrict us to one controller**
- **most apps have several controllers that each facilitate a relationship between a model and a set of views**
- you can think of an app as a **group of many different MVC implementations**



- **each screen that you see in the app** has an **associated model**, **an associated view** and a **controller** so that an *app with two screens* presenting *totally different information* could be *two sets of* **models**, **views** and **controllers**

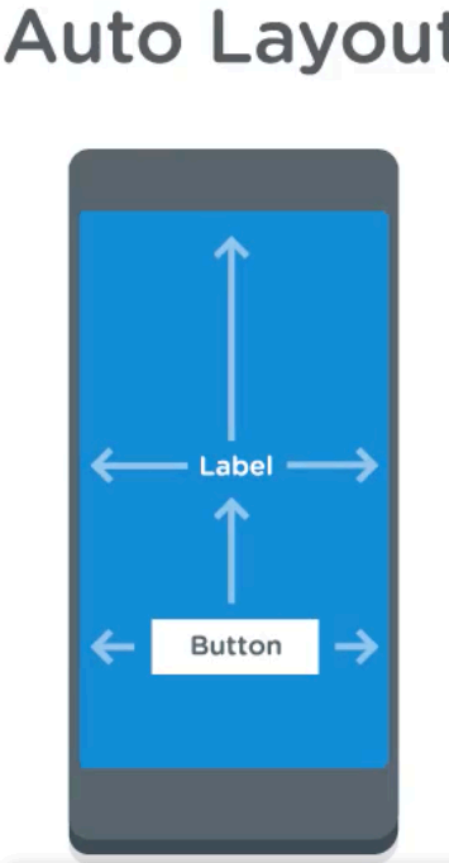


- why bother with the MVC pattern?
  * **seperating code into objects that do only one thing**
    ** means we **have to write more code**
  * it might not seem like it makes a difference right now but **as you write more complex apps**, **if all your code is in small number of files**
    ** **it becomes really difficult to figure out how an app works or what code causes errors when they arise**
- Apple considers the **MVC pattern** a **core competency** in *iOS development*
- **following good design patterns will help build good programming practices that become invaluable as you work with teams on larger projects**
- **following the MVC pattern** in *our project also builds on another* **good programming practice**, **the Single Responsibility Principle**
- **good object oriented design principle** states that **every class or object should have only a single responsibility**
  * and by **structuring our code** as **models**, **views** and **controllers**
    ** we **keep app responsibility narrowly within each class**
- you *should be aware* though that there's some **ambiguity when implementing the MVC design pattern** and *certain developers may do things slightly differently* than what you'll learn here
- **adhering to the MVC pattern is best learned through practice**
  * and as we learned, in this project and onwards, we will talk about the decisions of *where we implement things* and *how it relates to this pattern*
- you *should also note* that there are *slight variations on this pattern* that you may encounter across the web
  * you might read about **MVVM** or **model view, view model** or **MVP**, **model view presenter**

- for now, just **remember when writing your own code following a single responsibility principle and giving your objects very specific roles will help you write good code**

introduction to auto layout

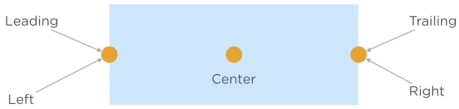
- to solve problems regarding layout in iOS
  * Apple has provided a system called **auto layout**
- using auto layout, we're not going to *specify the layout by inputting X or Y coordinates*
  * instead we **define the layout using mathematical relationship between the elements in our view**



- we can *define these relationships*, using either **constraints on individual elements** or **constraints between the set of elements**

    * once we've *redefined our layout using these relationships*, **our app can adapt to different layouts biased not only on device size** but on **orientation** and on **localisation** as well
- when using *auto layout* rather than *positioning our elements explicitly*, we **define constraints**
    * a **constraint** is essentially a **mathematical relationship between two elements**
- imagine a *text label* like the one we're using

- a **text label** has three *main constraint attributes*
    * horizontal
    * vertical
    * size
- for **horizontal positioning**, the *view* has a **leading constraint attribute**, a **trailing constraint attribute** as well as a *left*, *right* and *center*



- a **leading attribute** refers to the **edge where words and sentences begin**
- **trailing** refers to the **edge where words end**
    * there is a reason there is also *left and right* even though *trailing* and *leading* simply look like *left* and *right*
        ** this is because *not all languages follow the same direction*
- next we have **vertical positioning** and here we have *four possible constraints* that we can set
    * top
    * bottom
    * center
    * baseline

- the **baseline attribute** is available only for items that have a baseline
  * a *baseline* refers to an **invisible line offset from the bottom of an alignment rectangle** where **glifts of characters are laid out**

- we **typically pin a view** to its **super view**
- **tint colour** indicates **interactivity** and **selection state** for **UI elements**
  * while **text colour** is simply **stylised text**
- a **controller's job** is to **take data that is ready for presentation** and **give it to the view to present**
  * **creating and manipulating the data** is **not part of that role**
- you can **chain methods** in **one line** like so

```
let randomColor = ColorModel().getRandomColor()
```

advice

- a good workflow is whenever one can't find an answer
  * look into documentation first
  * then google
  * only then ask for help

adding an app icon

- an **icon** is essentially just a **set of images that add to our project**
- the system intelligently **displays the right image in the right context**
- Xcode lets us *manage the images* we use in our app through the use of an **asset catalog**

* the **asset catalog** is a **directory** that **can hold images** that you **use throughout your apps** but it *can also* **hold your app icons**