THE MODEL

Always has exactly four parts, although B) may be trivial

A) Fundamental data
   a. **Definition**: Information that is not stored anywhere else in your App. The authoritative source.
   b. **Implementation**: Must be stored, not computed, properties.
   c. **Note**: In the real world, it's not literally the source; it is usually initialized by persistent storage; see D). But as far as the View and Controller are concerned, it is the *only* source, and *obligatory* destination.
   d. **Representation**: Being the sole provider of fundamental data is not enough. A good representation is also critical. For example, storing prices as 'Float' or 'Double' can cause serious issues. Imagine a item that costs $0.69. As a Double, it is actually stored as **0.689999999999999467092948**.
   Doesn't seem like a big deal? Consider 2 algorithms that are mathematically identical:

```
let total = 100_000_000
let price = 0.69
// Algorithm 1
var amount: Double = 0.0
for i in 0..<total {
  amount += price
}
// Algorithm 2
var amount2 = Double(total) * price
println(NSString(format: "A1: %.20f A2: %.20f",
      amount, amount2))
```

Output:
A1: 68999999.90598693490028381348
A2: 69000000.00000000000000000000

If you were writing a high speed financial application, you would be quickly run out of town!

Similarly, imagine counting the mass of the observable universe in kilograms. Even a 64-bit unsigned integer, at a paltry 18 quintillion, or 1.8×10^{19} is still a quintillion quintillion times too small.

Finally, imagine storing the Oxford English Dictionary on your phone. Although it may be possible, it will probably use up many gigabytes of storage – a bad citizen. Instead, you should keep a smart cache (remember least/most frequently used?) of the 10,000 most commonly searched-for words, and make the rest available via a Web-based database request. (10,000 words is the average vocabulary of an educated adult.) However, when a Controller asks for a definition, it should not *appear* to happen any differently, maybe just slower.

    e. **Example**: A waffle costs $8.00. The customer just ordered 2. The fundamental data are 8.00, "waffle", and 2, stored & labeled clearly.

B) Derived data
    a. **Definition**: Values that can be calculated out of the fundamental data.
    b. **Implementation**: Often a computed property. May be a stored property if re-computing it is very expensive.
    c. **Example**: the subtotal is $16.00. The *knowledge* of how to calculate the derived data *must still reside strictly within the model, not the View nor the controller.* The formula "subtotal = pricePerWaffle * countOfWaffles" must not appear anywhere else.

C) A notification system
    a. **Definition**: A method of broadcasting changes in the model to all interested parties.
    b. **Usage note**: A method must broadcast any change which might possibly affect any view of its data. A "view" here is in the most general sense; a view could be a sound recording, or a database located on a remote Internet host. So the model must not make any assumptions about the type and number of interested views.

D) A method of initializing itself.
    a. Definition: A model must start off in a known, consistent state.
    b. **Examples**: To do so, it may use any number of sources:
        i. A hardcoded Array/Dictionary in a .swift file (least preferable)
        ii. The User preference system provided by iOS
        iii. The Application's storage "sandbox" in the device filesystem
        iv. The SQL database in the device, usually interfaced through CoreData library
        v. A "plist" (property list) file written by the developer and imported via Xcode
        vi. Somewhere on the Internet, usually with Web (HTTP)/JSON interface to a Server database, but others are possible, like REST.

THE VIEW

Definition: A hierarchy of (possibly interactive) display elements that are supplied by the operating environment and customized by the App's developers and designers. Provides the visual and touch interaction with the user.

Implementation: In iOS, the screen contains a root window (a UIWindow) which has a single child (a UIView) which fills the screen. In turn, this parent UIView contains any number of children views, which may themselves contain children, etc. All "widgets" (visual, possibly interactive elements) are instances of UIView or derive from UIView. A UIBezierPath is not a view; it just paints pixels onto a view.

Appearance: Views must know how they should appear. They delegate as follows:

1) For cosmetic aspects such as font, color, line thickness, left/right justification, line wrap, and layout constraints, they are specified by Interface Builder*.
2) For simple data values such as a UILabel's or UITextField's or UITextView's initial value, they are also specified by Interface Builder.
3) For anything more complex such as Table Views, and Collection Views, and custom derived classes of UIView, they **must delegate to a model. This model must be known as an abstract type, that is, a Protocol. This takes thought.**

Design: *UIViews are fundamentally passive.* Views are nothing but a rendering algorithm. Rendering is complicated and makes heavy use of Core Graphics. An example graphics-related calculation is figuring out where the pixels should go to approximate a circle. This would involve trigonometry.

Views rely on iOS to tell them when to "repaint" themselves; they rely entirely on IB attributes and delegates to supply the data they should contain. The rendering algorithm plus the supplied data determines their final appearance.

Any delegates *must be managed by the Controller and supplied by the Controller.* In simple cases a Controller may assume delegate responsibilities directly. Otherwise the Controller instantiates and supplies a separate model instance to its views. A view must not assume or supply its own delegate.

Design Exception: The simplest UIViews use a shortcut: a UITextfield in fact has its model internally: the 'text' property; the same goes for a UIStepper which has an internal 'value' property. I feel that this lack of purity causes confusion and tempts programmers to "stuff" their model within their own custom UIView. Although, purity would create a high bar for novice programmers. In any case, study UIPickers, and UITableViews, not UITextfields, to understand the fundamental patterns.

*Not strictly speaking. It is possible to do anything done in Interface Builder in code or by hand editing XML. Dynamically setting IB attributes is sometimes necessary. The fundamentals are hard. Keep these exceptions for later.

CONTROLLERS (UIViewController)

Controllers are the heart of your application and where you spend a great deal of your time coding. They literally "pull the whole thing together."

Definition: The UIViewController receives all messages from iOS relating to the lifecycle of the current screen, or UIView, of an App. These messages constitute life-cycle of that ViewController. A ViewController's lifecycle starts when it appears on screen, and ends when it is dismissed from view.

LifeCycle: ViewControllers come into and out of being based on the navigation you've designed in your App. We haven't covered navigation yet. For now, with Single-View apps, iOS brings a single ViewController to life when the App starts, and tears it down when the App terminates.

(An App terminates only due to: crashing; out-of-memory errors; other misbehavior such as excessive CPU usage; device reboot; or user-forced termination. Users prefer to keep Apps as they were for convenient resumption at any time. Hence your root ViewController will be quite persistent.)

VIEW CONTROLLER RESPONSIBILITIES

A) When the ViewController's life starts, it's responsible for gaining access to all models needed by all of its child views. [These models may be instantiated directly, such as in viewDidLoad(), or pulled from an App-wide repository. The latter case may be more efficient if instantiating the model is time consuming and multiple views need access to it. This involves engineering & forethought.]

B) Once its models are initialized, it must initialize the delegates of its child views, so that child views paint themselves correctly when asked to do so. This is just careful housekeeping.

C) Also once its models are initialized, it must listen to all relevant broadcasts from these models. There must be a good convention for listening correctly, by avoiding any "magic constants". This is just a matter of keeping good habits.

D) As @IBActions occur, it must translate low-level iOS Events (touches; drags; taps, pinches; zooms; long-taps) into App-level events (item added to restaurant bill; user wishes to pay bill). An App-level event means either or both of:

   a. Update the model (then, see E.)
   b. Navigate away from the current view (we'll get to that later)

E) As broadcast messages from its models arrive, the controller must use its internal knowledge of its child view behavior to pass on re-painting messages to the right views. This is another place where real thinking/design comes in.

F) When the ViewController's life ends, it's responsible for releasing any persistent resources such as timers, images, and observers. This is just more good housekeeping.

Your apps, especially your final project, will be graded with respect to complying with this schema. So, please ask if you're not sure where a line of code should go!