iOS Foundations II
Day 7

- Custom Table View Cell
- Dictionaries
- Property List
- Optional Binding

Custom UITableViewCell

TableViewCell

- UITableViewCell is a direct subclass of UIView.
- Can be considered a regular view that contains a few other views used to display information.
- The ‘Content View’ of a cell is the view that all content of a table view cell is placed on. Consider it as the default superview of your cell. ContentView itself is a read only property; you can’t set it to be a different view.

TableViewCell Style

- Setting the **style** of a UITableViewCell object exposes certain interface objects of the cell.
- The **Default** style exposes the default text label and optional image view.
- **Right Detail** and **Left Detail** expose a right (left)-aligned detail text label on the right (left) side of the cell in addition to the default text label.
- **Subtitle** exposes a left aligned label below the default text label.

Creating TableView Cells

- You can instantiate them in code with the initializer init(style: UITableViewCellStyle, reuseIdentifier: String?)
- But usually you'll set them up in your storyboard or in a xib file.
- If they're in your storyboard, you just need to set their reuse identifier in the identity inspector, and then call dequeueReusableCellWithIdentifier() at the appropriate time.

Custom UITableViewCell

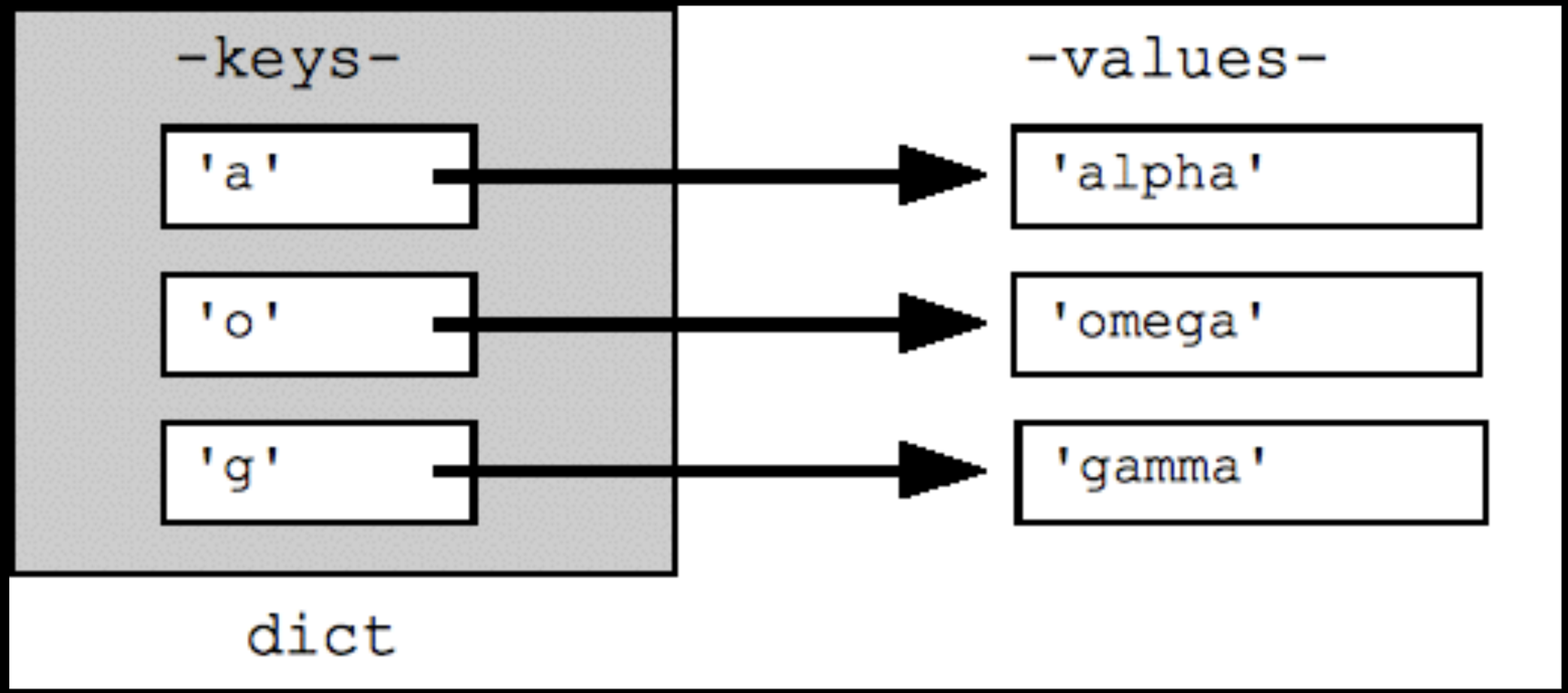
- Creating and laying out your own custom UITableViewCell is a relatively simple workflow:
    1. Create a new class that is a subclass of UITableViewCell
    2. In your storyboard, set your prototype tableview cell to be your new custom class
    3. Drag out any interface elements you want onto your prototype cell
    4. Create outlets to each element in your custom class's implementation
    5. Refactor your cellForRow:AtIndexPath: method on your tableview's datasource

Demo 7.1:
Table view w/ custom cells & data as object properties

Dictionaries

- Dictionaries are a collection type, like array.
- A dictionary keeps track of things in key-value pairings, as apposed to an array that keeps an index. Dictionaries are unordered in that regard.
- You store an object in a dictionary by calling setObject:ForKey:
- But you almost never use that long hand form, instead use the [] shorthand syntax, just like an array
- When you need to retrieve the object from the dictionary, you can call objectForKey: and provide the original key you set it with (use the short hand form instead)
- Dictionaries do not allow duplicate keys.

Dictionaries



Creating Dictionaries

- Just like with arrays, there is a literal syntax for creating Dictionaries:

```
var info = ["Year" : 2014]
```

- Setting a value in a dictionary also has a special shorthand syntax:

```
info["Month"] = 11
info["Day"] = 17
```

- As well as accessing a value:

```
var today = info["Day"]
```

Demo 7.2: Dictionaries

plist (property list)

- Apple-flavored XML
- Keys must be strings
- values must be NSCoding compliant
- Load from your bundle or from the web
- Root-level object is typically a Dictionary or Array

Xcode GUI format

Key	Type	Value
▼ Root	Array	(3 items)
▼ Item 0	Dictionary	(2 items)
FirstName	String	Jimmy
LastName	String	Graham
▼ Item 1	Dictionary	(2 items)
LastName	String	Wilson
FirstName	String	Russell
▼ Item 2	Dictionary	(2 items)
LastName	String	Johnson
FirstName	String	Brad

Raw format

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>FirstName</key>
    <string>Jimmy</string>
    <key>LastName</key>
    <string>Graham</string>
  </dict>
  <dict>
    <key>LastName</key>
    <string>Wilson</string>
    <key>FirstName</key>
    <string>Russell</string>
  </dict>
  <dict>
    <key>LastName</key>
    <string>Johnson</string>
    <key>FirstName</key>
    <string>Brad</string>
  </dict>
</array>
</plist>
```

plist Workflow

1. Get the URL of the plist file living in your bundle
2. Read the plist into an Array or Dictionary using class methods on the Objective-C collection types, based on the root object of the plist
3. Parse the data in the plist into your model objects

Demo 7.3:
Table view w/ custom cells & data as a plist

Optional Binding

- You can use **optional binding** to find out wether an optional contains a value, and if so, to make that value available as a temporary constant or variable that is unwrapped.
- The syntax of an optional binding:

```
if let constantName = someOptional {
    statements
}
```

using optional binding

```
func printTitleValue(value : String?) {

    if let title = value {
        println(title)
    }
}
```

not using optional binding

```
func printTitleValue(value : String?) {

    if value != nil {
        let title = value!
        println(title)
    }
}
```

Downcasting + Optional Binding

- It is very common to combine optional binding with **downcasting**
- Downcasting is used whenever a constant or variable of a certain type may actually refer to an instance of a subclass behind the scenes.
- When you think this is the case, you can use downcasting to attempt to cast the variable or constant to the subclass.
- There are two forms of down casting:
  - optional form: as?
  - forced form : as!

Demo 7.4:
Optional binding
& downcasting