iOS Foundations II
Day 7

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

Custom UITableViewCell

TableViewCell

- UITableViewCell is a direct subclass of UIView.
- You can think of it as a regular view that contains a number of other views used to display information.
- The ‘Content View’ of a cell is the view that all content of a table view cell should be placed on. Think of it as the default super view of your cell. contentView itself is read only, you cant set it to be a different view.

TableViewCell Style

- Setting the style of an instance of UITableViewCell will expose certain interface objects on the cell.
- The default style exposes the default text label and optional image view.
- Right Detail exposes a right aligned detail text label on the right side of the cell in addition to the default text label.
- Left Detail exposes a left aligned detail text label on the right 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 will be setting them up in your storyboard or in a xib file.
- If they are in your storyboard, you just have to set their reuse identifier in the identity inspector, and then call dequeueReusableCellWithIdentifier() at the appropriate time.

Custom UITableViewCell

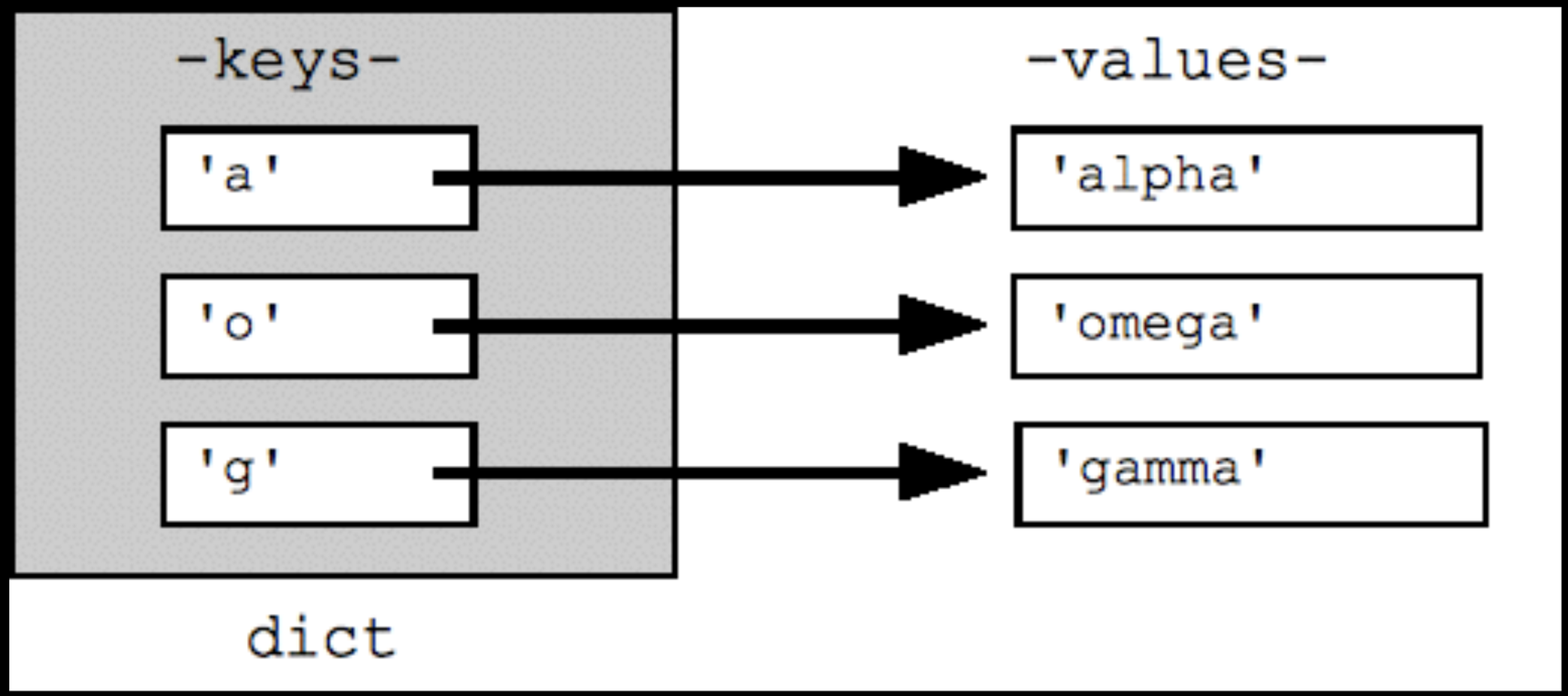
- Creating and laying out your own custom UITableViewCell is a relatively simple workflow:
    1. Create a new class that is a subclass of UITableViewCell
    2. In your storyboard, set your prototype tableview cell to be your new custom class
    3. Drag out any interface elements you want onto your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

Demo

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:
- When you need to retrieve the object from the dictionary, you can call objectForKey: and provide the original key you set it with.
- 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

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

plist Workflow

1. Get the URL of the plist file
2. Read the plist into an Array or Dictionary using class methods on those collection types
3. Use optional binding & down casting to parse the data and use objectForKey: (only on dictionaries) or subscripting: array[n], dictionary[key]
4. Use a for loop to parse a collection into your model objects

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

- So we can combine optional binding and downcasting to achieve those checks in one line.
- 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

plist Workflow again!

1. Get the URL of the plist file
2. Read the plist into an Array or Dictionary using class methods on those collection types
3. Use conditional unwrapping to parse the data and use objectForKey: (only on dictionaries) or subscripting: array[n], dictionary[key]
4. Use a for loop to parse a collection into your model objects

plist sample code

```
func loadFromPList() {
    //get the path to our plist
    let path = NSBundle.mainBundle().pathForResource("roster", ofType: "plist")
    //create an array from the plist file, if the root of the plist was a dictioanry instead of an
   array we would call the same method but on NSDictionary
    let plistArray = NSArray(contentsOfFile: path!)
    //loop through the dictionaries in our plist array
    for object in plistArray! {
        //optional
        if let personDictionary = object as? NSDictionary {
        //gotta cast these because Xcode isnt sure what kind of values you will get out of the
     dictionaries
        let firstNameFromPlist = personDictionary["firstName"] as String
        let lastNameFromPlist = personDictionary["lastName"] as String
        let newPerson = Person(first: firstNameFromPlist, last: lastNameFromPlist)
        //add them to our array
        self.people.append(newPerson)
        }
    }
}
```

Demo