PUSHING THE LIMITS OF PROTOCOL-ORIENTED PROGRAMMING

JESSE SQUIRES

jessesquires.com • @jesse_squires

THE SWIFT WAY

How do we write swifty code?

Standard library:

"Protocols and value types all the way down."

WRITING "SWIFTY" CODE MEANS WRITING PROTOCOL-ORIENTED CODE

WHAT IS PROTOCOL-ORIENTED PROGRAMMING?

SOLID

DESIGN PRINCIPLES

SOLID

INTERFACE SEGREGATION

"PROTOCOL-ORIENTED"
INTERFACE SEGREGATION

WAT?

What do we mean by
"interface segregation"?

PROTOCOLS

The "I" in SOLID:
No client should depend on (or know about) methods it does not use.

» Small, separated interfaces == focused API

» Restrict access

» Unify disjoint types

» Hide concrete types

WHY?

“Why do we want to use protocols?”

— Greg Heo

PROTOCOLS HELP US WRITE CODE THAT IS:

1. modular
2. dynamic
3. testable

WHAT IF EVERYTHING WERE A protocol?

(well, almost)

Let's find out!

EXPERIMENT:
LET'S BUILD PROTOCOL-ORIENTED DATA SOURCES

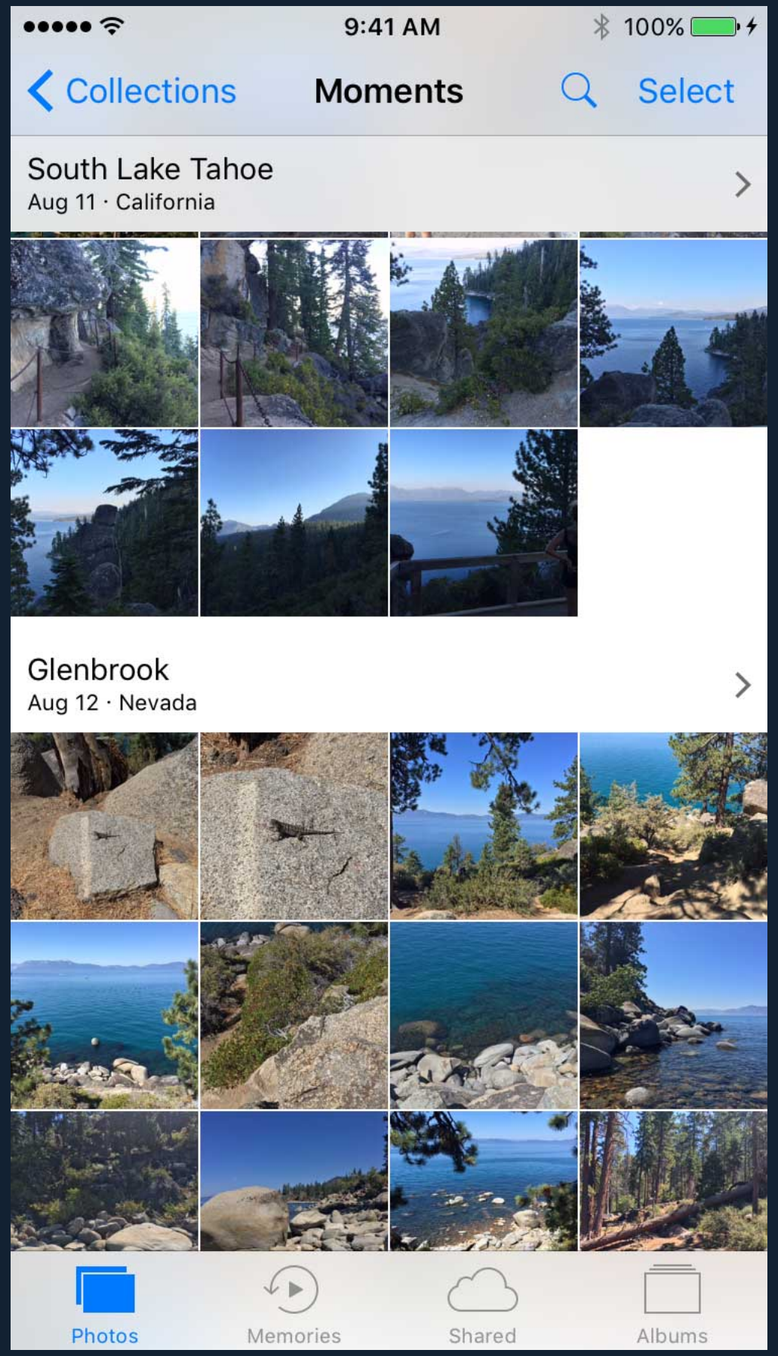
- `UITableViewDataSource`
- `UICollectionViewDataSource`

Table views and collection views are fundamental components of most apps.

GOALS

1. protocol-based
2. type-safe / generic
3. unify UITableView and UICollectionView
4. remove UIKit boilerplate
5. avoid NSObject and NSObjectProtocol ("pure" Swift)


9:41 AM
100%
Settings
General
About
Software Update
Spotlight Search
Handoff
CarPlay
Accessibility
Storage & iCloud Usage
Background App Refresh
Restrictions
Off


9:41 AM
100%
Collections
Moments
Select
South Lake Tahoe
Aug 11 · California
Glenbrook
Aug 12 · Nevada
Photos
Memories
Shared
Albums

RESPONSIBILITIES

Display data in a list or grid.

What do we need?

1. Structured data (sections with items/rows)
2. Create and configure cells
3. Conform to UITableViewDataSource
4. Conform to UICollectionViewDataSource

SECTION PROTOCOL

```
protocol SectionInfoProtocol {
    associatedtype Item

    var items: [Item] { get set }

    var headerTitle: String? { get }

    var footerTitle: String? { get }
}
```

SECTION TYPE

```
struct Section<Item>: SectionInfoProtocol {

    var items: [Item]

    let headerTitle: String?

    let footerTitle: String?
}
```

DATASOURCE PROTOCOL

```
protocol DataSourceProtocol {
    associatedtype Item

    func numberOfSections() -> Int

    func numberOfItems(inSection section: Int) -> Int

    func item(atRow row: Int, inSection section: Int) -> Item?

    func headerTitle(inSection section: Int) -> String?

    func footerTitle(inSection section: Int) -> String?
}
```

DATASOURCE TYPE

```
struct DataSource<S: SectionInfoProtocol>: DataSourceProtocol {

    var sections: [S]

    // MARK: DataSourceProtocol

    func numberOfSections() -> Int {
        return sections.count
    }

    // other protocol methods...
}
```

RESPONSIBILITIES

✅ Structured data

2. Create and configure cells

3. Conform to UITableViewDataSource

4. Conform to UICollectionViewDataSource

CREATE + CONFIGURE CELLS

1) We need a common interface (protocol) for:

- Tables & collections
- Table cells
- Collection cells

2) We need a unified way to create + configure cells

UNIFY TABLES + COLLECTIONS

```
// UITableView
// UICollectionView

protocol CellParentViewProtocol {
    associatedtype CellType: UIView

    func dequeueReusableCellFor(identifier: String,
                                indexPath: IndexPath) -> CellType
}
```

UNIFY TABLES + COLLECTIONS

```
// conform collection view
// (table view is similar)

extension UICollectionView: CellParentViewProtocol {
    typealias CellType = UICollectionViewCell

    func dequeueReusableCellFor(identifier: String,
                                indexPath: IndexPath) -> CellType {
        return dequeueReusableCell(withReuseIdentifier: identifier,
                                   for: indexPath)
    }
}
```

UNIFY CELLS

```
// UITableViewCell
// UICollectionViewCell

protocol ReusableViewProtocol {
    associatedtype ParentView: UIView, CellParentViewProtocol

    var reuseIdentifier: String? { get }

    func prepareForReuse()
}
```

UNIFY CELLS

```
extension UICollectionViewCell: ReusableViewProtocol {
    typealias ParentView = UICollectionView
}

extension UITableViewCell: ReusableViewProtocol {
    typealias ParentView = UITableView
}

// already implemented in UIKit
//
// var reuseIdentifier: String? { get }
// func prepareForReuse()
```

SHARING A COMMON INTERFACE

CellParentViewProtocol

» UITableView and UICollectionView

ReusableViewProtocol

» UITableViewCell and UICollectionViewCell

CREATE + CONFIGURE CELLS

✅ A common interface

2. A unified way to create and configure cells

CONFIGURE CELLS PROTOCOL

```
// configure a cell with a model

protocol ReusableViewConfigProtocol {
    associatedtype Item
    associatedtype View: ReusableViewProtocol

    func reuseIdentiferFor(item: Item?,
                           indexPath: IndexPath) -> String

    func configure(view: View,
                   item: Item?,
                   parentView: View.ParentView,
                   indexPath: IndexPath) -> View
}
```

CREATE CELLS EXTENSION

```
// for table view, similar for collection view
extension ReusableViewConfigProtocol where View: UITableViewCell {
    func tableCellFor(item: Item,
                      tableView: UITableView,
                      indexPath: IndexPath) -> View {

        let cellId = self.reuseIdentiferFor(item: item, indexPath: indexPath)

        // CellParentViewProtocol
        let cell = tableView.dequeueReusableCellFor(identifier: cellId,
                                                    indexPath: indexPath) as! View

        return self.configure(view: cell,
                              item: item,
                              parentView: tableView,
                              indexPath: indexPath)
    }
}
```

CREATE + CONFIGURE CELLS TYPE

```
struct ViewConfig<Item, Cell: ReusableViewProtocol>: ReusableViewConfigProtocol  {
    let reuseId: String
    let configClosure: (Cell, Item, Cell.ParentView, IndexPath) -> Cell

    // ReusableViewConfigProtocol

    func reuseIdentiferFor(item: Item?,
                           indexPath: IndexPath) -> String {
        return reuseId
    }

    func configure(view: View,
                   item: Item?,
                   parentView: View.ParentView,
                   indexPath: IndexPath) -> View {
        return configClosure(view, item, parentView, indexPath)
    }
}
```

RESPONSIBILITIES

✅ Structured data

✅ Create and configure cells

3. Conform to UITableViewDataSource

4. Conform to UICollectionViewDataSource

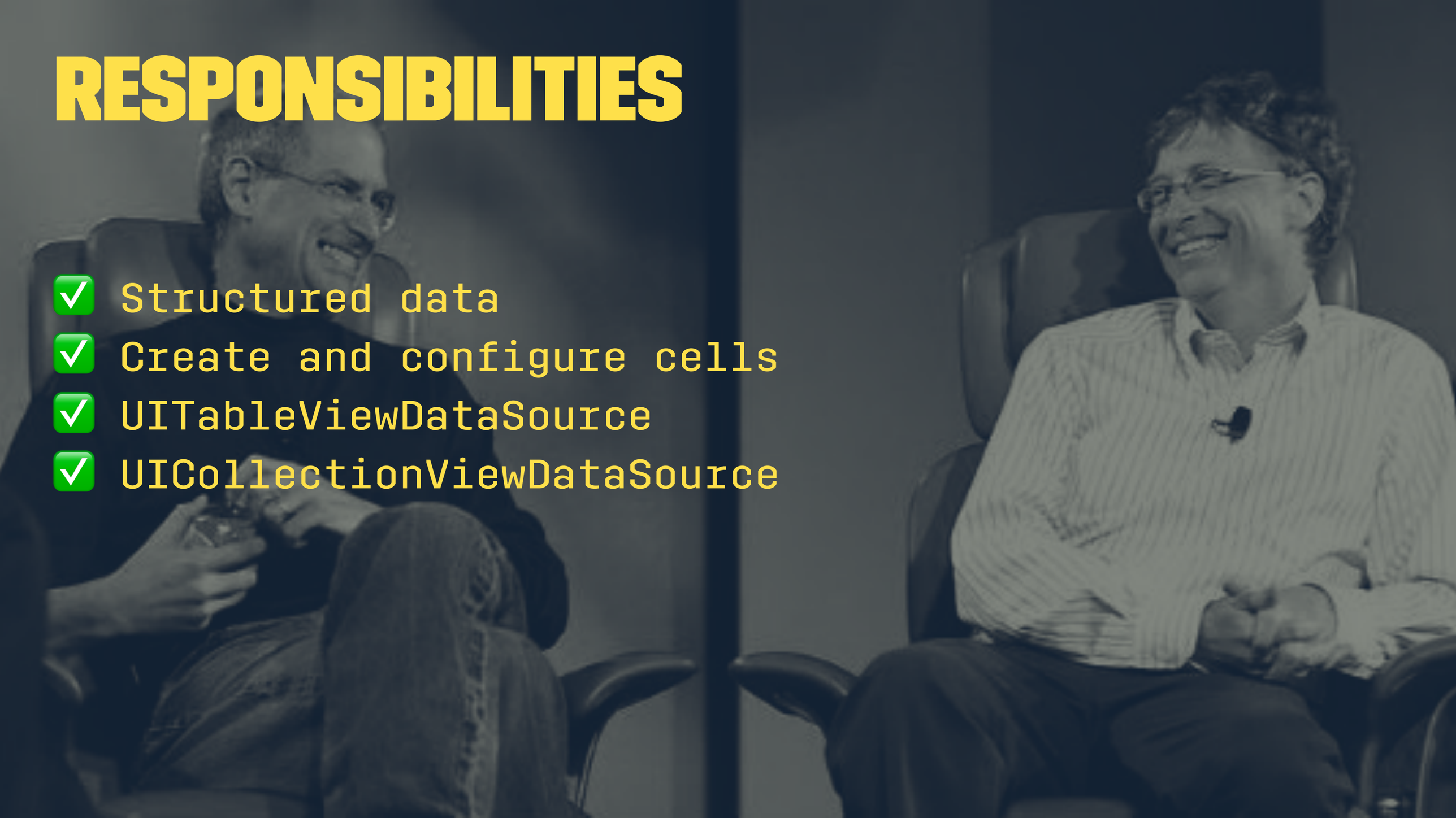
DATA SOURCE PROTOCOLS

```
class BridgedDataSource: NSObject,
                         UICollectionViewDataSource,
                         UITableViewDataSource {

    // Init with closures for each data source method

    // Implement UICollectionViewDataSource

    // Implement UITableViewDataSource
}
```

DATA SOURCE PROTOCOLS EXAMPLE

```
class BridgedDataSource: NSObject {

    let numberOfSections: () -> Int

    // other closure properties...
}

extension BridgedDataSource: UICollectionViewDataSource {

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return self.numberOfSections()
    }

    // other data source methods...
}
```

RESPONSIBILITIES



- ✅ Structured data
- ✅ Create and configure cells
- ✅ UITableViewDataSource
- ✅ UICollectionViewDataSource

EVERYTHING WE NEED

```
protocol SectionInfoProtocol { } // sections of items

protocol DataSourceProtocol { } // full data source

protocol CellParentViewProtocol { } // tables + collections

protocol ReusableViewProtocol { } // cells

protocol ReusableViewConfigProtocol { } // configure cells

class BridgedDataSource { } // UIKit data sources
```

CONNECTING THE PIECES

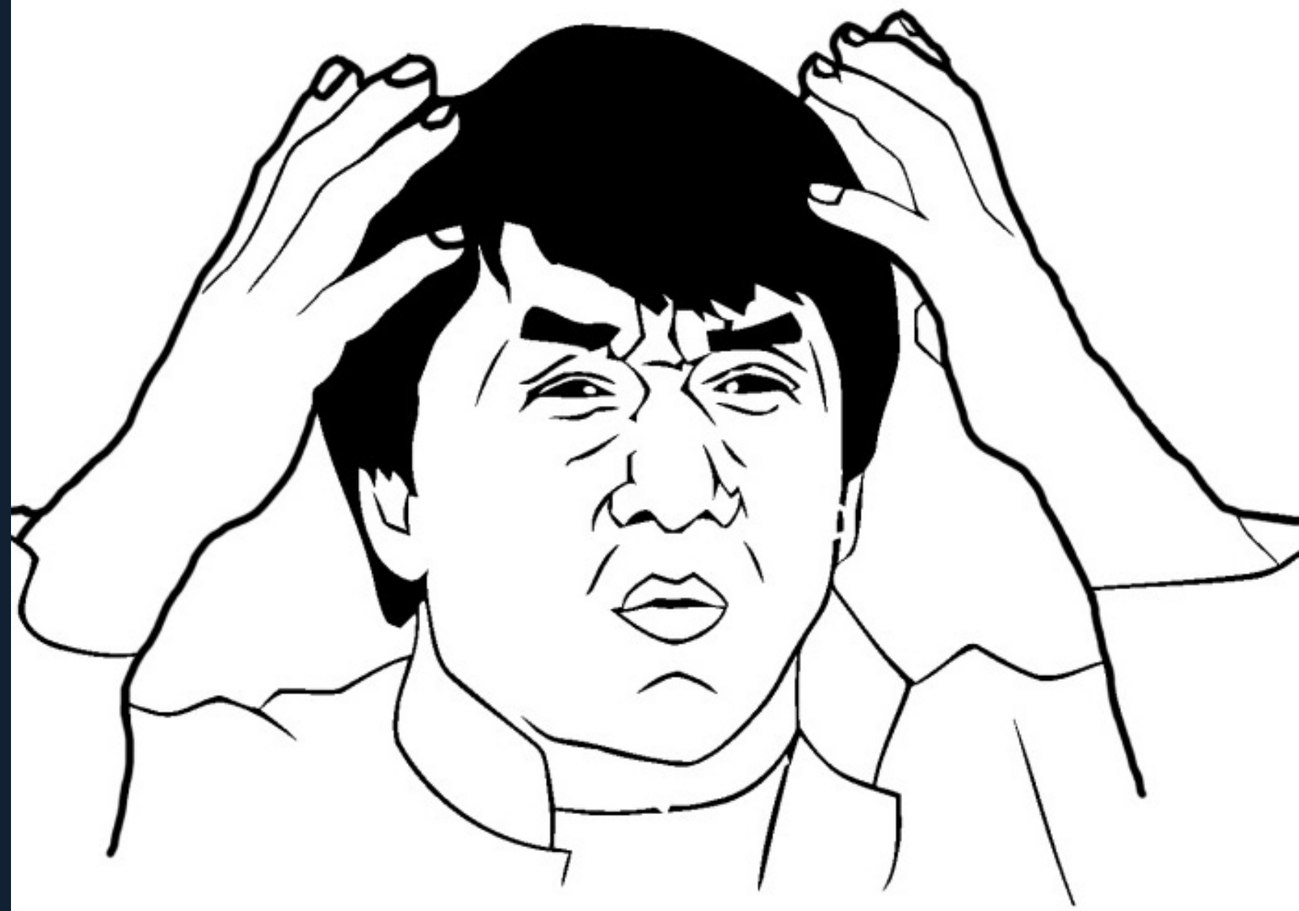
```
class DataSourceProvider<D: DataSourceProtocol,
                         C: ReusableViewConfigProtocol>
      where D.Item == C.Item {

    var dataSource: D
    let cellConfig: C

    private var bridgedDataSource: BridgedDataSource?

    init(dataSource: D, cellConfig: C)
}
```

RESULTS

```
let data = DataSource(sections: /* sections of models */)

let config = ViewConfig(reuseId: "cellId") { (cell, model, view, indexPath) -> MyCell in
    // configure cell with model
    return cell
}

let provider = DataSourceProvider(dataSource: data, cellConfig: config)

// connect to collection
collectionView.dataSource = provider.collectionViewDataSource

// connect to table
tableView.dataSource = provider.tableViewDataSource
```

HOW DOES THAT WORK?

```
// Collections
provider.collectionViewDataSource

// Tables
provider.tableViewDataSource
```



GENERATING SPECIFIC DATA SOURCES

```
// class DataSourceProvider<D,C>
// private var bridgedDataSource: BridgedDataSource?

extension DataSourceProvider where C.View: UITableViewCell {

    public var tableViewDataSource: UITableViewDataSource {
        // create and return new BridgedDataSource
        // using self.dataSource and self.cellConfig
    }

    private func createTableViewDataSource() -> BridgedDataSource {
        // ...
    }
}
```

GENERATING SPECIFIC DATA SOURCES

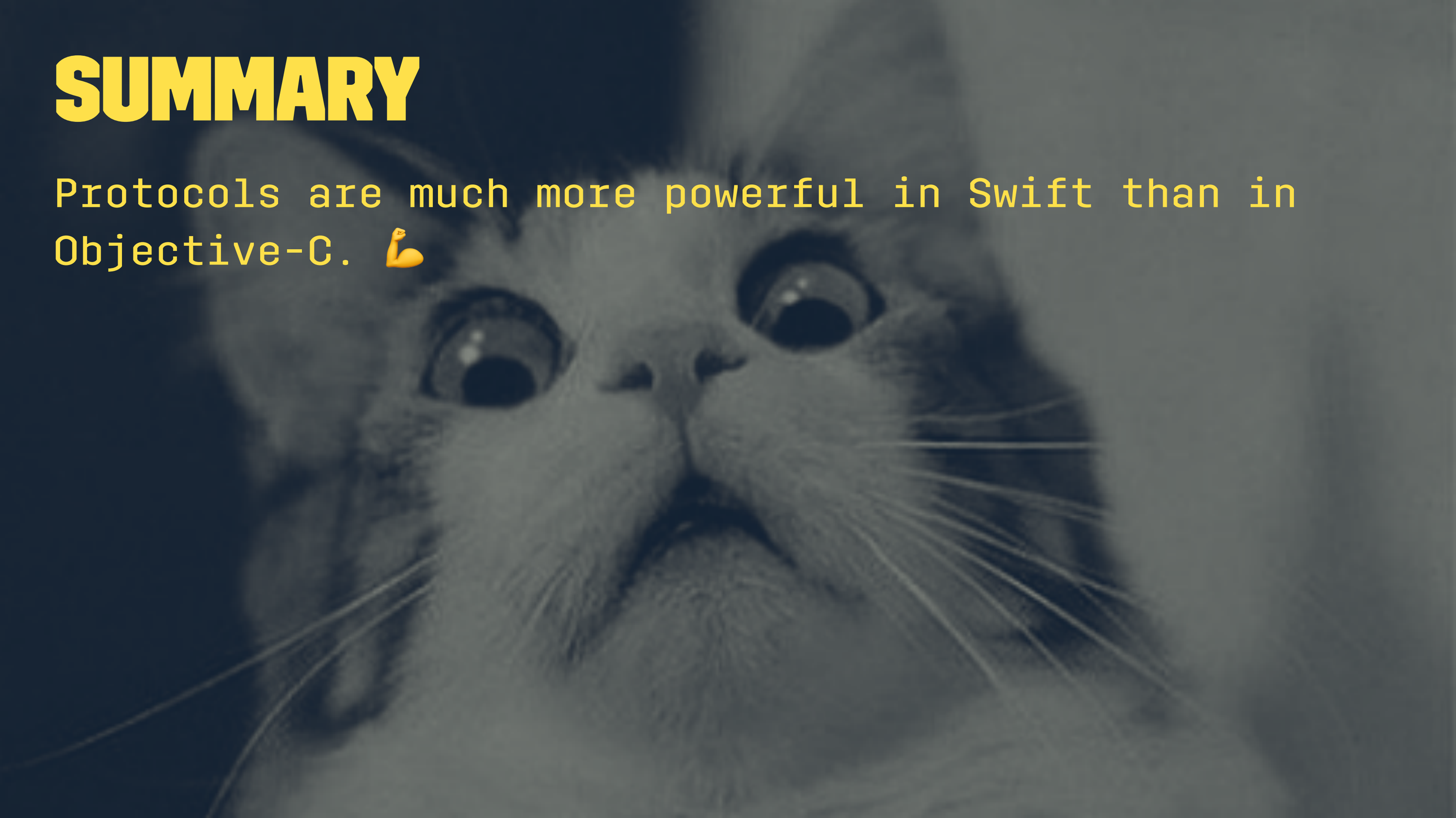
```
// extension DataSourceProvider where C.View: UITableViewCell
func createTableViewDataSource() -> BridgedDataSource {
    let source = BridgedDataSource()

    source.numberOfSections = { () -> Int in
        return self.dataSource.numberOfSections()
    }

    source.numberOfItemsInSection = { (section) -> Int in
        return self.dataSource.numberOfItems(inSection: section)
    }

    source.tableCellForRowAtIndexPath = { (tableView, indexPath) -> UITableViewCell in
        let item = self.dataSource.item(at: indexPath)
        // extension method on ReusableViewConfigProtocol
        return self.cellConfig.tableCellFor(item: item,
                                            tableView: tableView,
                                            indexPath: indexPath)
    }
}
```

RESULTS — ONE MORE TIME

```
let data = DataSource(sections: /* sections of models */)

let config = ViewConfig(reuseId: "cellId") { (cell, model, view, indexPath) -> MyCell in
    // configure cell with model
    return cell
}

let provider = DataSourceProvider(dataSource: data, cellConfig: config)

// connect to collection
collectionView.dataSource = provider.collectionViewDataSource

// connect to table
tableView.dataSource = provider.tableViewDataSource
```

SUMMARY

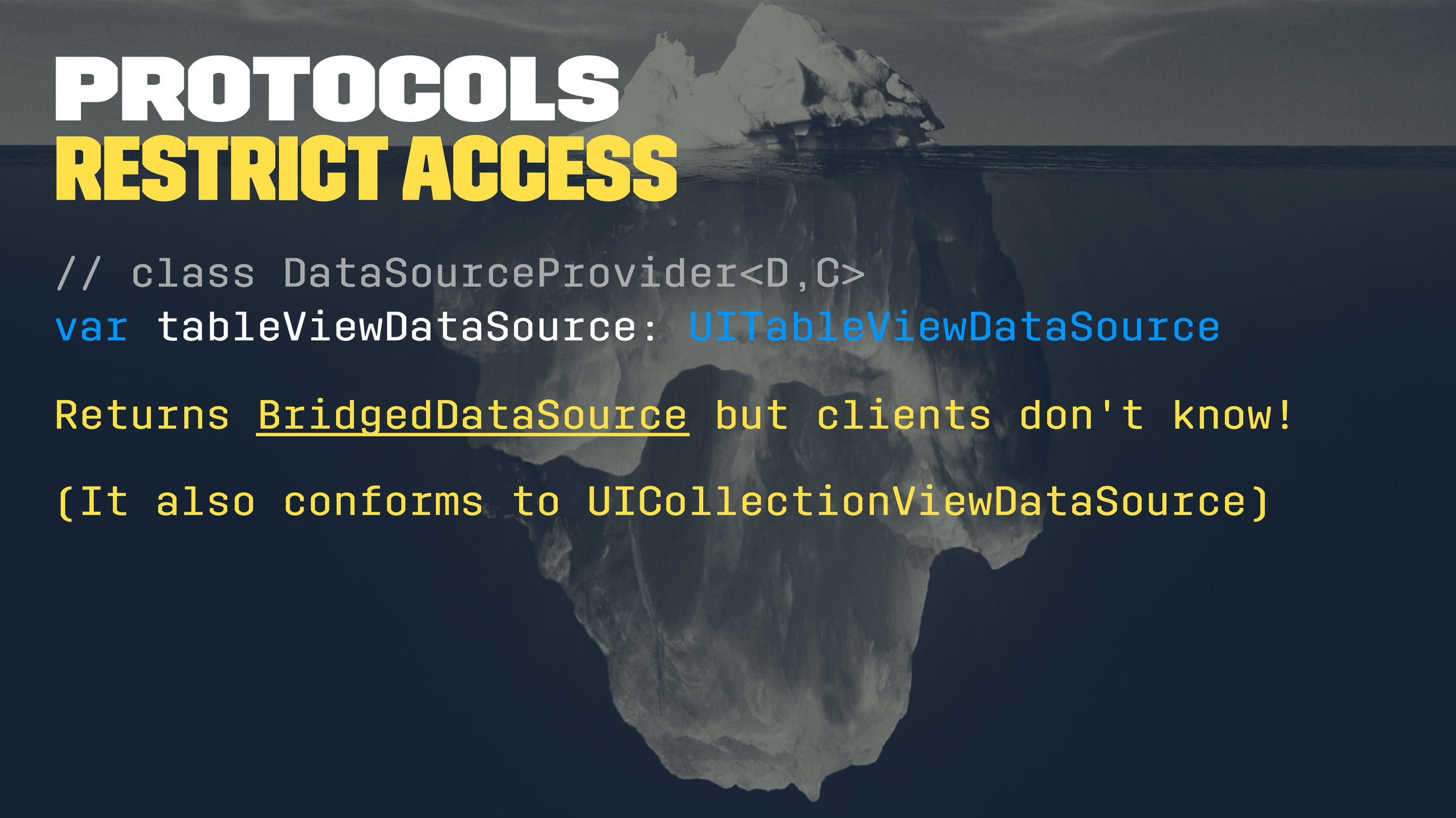
Protocols are much more powerful in Swift than in Objective-C. 💪



PROTOCOL EXTENSIONS
DYNAMIC INTERFACE SEGREGATION

```
extension ReusableViewConfigProtocol where View: UITableViewCell {
    func tableCellFor(item: Item,
                      tableView: UITableView,
                      indexPath: IndexPath) -> View
}

extension DataSourceProvider where C.View: UITableViewCell {
    var tableViewDataSource: UITableViewDataSource
}
```

You cannot access tableViewDataSource if you are creating UICollectionViewCells!



PROTOCOLS RESTRICT ACCESS

```
// class DataSourceProvider<D,C>
var tableViewDataSource: UITableViewDataSource
```

Returns BridgedDataSource but clients don't know!

(It also conforms to UICollectionViewDataSource)

PROTOCOLS
UNIFY DISJOINT TYPES
HIDE TYPES

```
protocol CellParentViewProtocol { }

protocol ReusableViewProtocol { }
```

We can treat tables, collections and their cells the same way — speaking to the same interface.

PROTOCOLS
MODULAR

```
protocol SectionInfoProtocol { }

protocol DataSourceProtocol { }

protocol ReusableViewConfigProtocol { }
```

Anything can be a data source

Anything can configure cells

PROTOCOLS
TESTABLE

Easy to "mock" or fake
a protocol in a unit test.

Easy to verify that
a protocol method was called.

PROTOCOLS + EXTENSIONS EXPAND OUR DESIGN SPACE

THANKS!

Me:
@jesse_squires
jessesquires.com

Swift Weekly Brief:
@swiftlybrief
swiftweekly.github.io