iOS Application Development

Lecture 9: Advanced Data Display:
Swift Generics • Dynamic Data • Compositional Layout

Prof. Dr. Jan Borchers
Media Computing Group
RWTH Aachen University

WS ’22/’23 • hci.rwth-aachen.de/ios
Swift Generics
let ints = [1, 2, 3, 4]
let strings = ["1", "2", "3", "4"]

var ints: [Int]
var strings: [String]

var ints: Array<Int>
var strings: Array<String>

struct Array<Element> {
}

Prof. Dr. Jan Borchers: iOS Application Development
let wordsByLength = [
  1: ["a", "i"],
  2: ["hi", "by", "go"],
  3: ["the", "but", "now", "how"],
  4: ["then", "just", "when", "cool"]
]

var wordsByLength: [Int: [String]]

var wordsByLength: Dictionary<Int, [String]>

struct Dictionary<Key, Value> where Key : Hashable {
}

Dictionary
Type Constraints

```swift
struct Dictionary<Key, Value> where Key: Hashable {
}

struct Dictionary<Key: Hashable, Value> {
}
```
Functions and Methods

func max(_ x: Int, _ y: Int) -> Int {
}

func max<T>(_ x: T, _ y: T) -> T where T : Comparable {
}

func max<T>(_ x: T, _ y: T) -> T where T : Comparable {
    if y >= x {
        return y
    } else {
        return x
    }
}
Associated Types

- What about protocols?
- Associated types enable protocols to define placeholders for any type
- Enables “generic protocols”

```swift
protocol APIRequest {
    associatedtype Response

    var urlRequest: URLRequest { get }
    func decodeResponse(data: Data) throws -> Response
}
```
Associated Types

```swift
protocol APIRequest {
    associatedtype Response

    var urlRequest: URLRequest { get }
    func decodeResponse(data: Data) throws -> Response
}

func sendRequest_<Request: APIRequest>(request: Request) async throws -> Request.Response {
    let (data, response) = try await URLSession.shared.data(for: request.urlRequest)

    guard let httpResponse = response as? HTTPURLResponse,
    httpResponse.statusCode == 200 else {
        throw APIRequestError.itemNotFound
    }

    let decodedResponse = try request.decodeResponse(data: data)
    return(decodedResponse)
}
```
struct PhotoInfoAPIRequest: APIRequest {
    var apiKey: String

    var urlRequest: URLRequest {
        var urlComponents = URLComponents(string: "https://api.nasa.gov/planetary/apod")!
        urlComponents.queryItems = [URLQueryItem(name: "api_key", value: apiKey)]
        return URLRequest(url: urlComponents.url!)
    }

    func decodeResponse(data: Data) throws -> PhotoInfo {
        let photoInfo = try JSONDecoder().decode(PhotoInfo.self, from: data)
        return photoInfo
    }
}

let photoInfoRequest = PhotoInfoAPIRequest(apiKey: "DEMO_KEY")
let result = try await sendRequest(photoInfoRequest)
// handle result
struct ImageAPIRequest: APIRequest {
    enum ResponseError: Error {
        case invalidImageData
    }
    let url: URL
    var urlRequest: URLRequest {
        return URLRequest(url: url)
    }

    func decodeResponse(data: Data) throws -> UIImage {
        guard let image = UIImage(data: data) else {
            throw ResponseError.invalidImageData
        }
        return image
    }
}

let photoInfoRequest = PhotoInfoAPIRequest(
    apiKey: "DEMO_KEY"
)

Task {
    do {
        let photoInfo = try await sendRequest(photoInfoRequest)
        print(photoInfo)
        let imageRequest = ImageAPIRequest(
            url: photoInfo.url)
        let image = try await sendRequest(imageRequest)
        image
    } catch {
        print(error)
    }
}
Dynamic Data
Search Controllers

• Used to provide search functionality

• Integrates directly with navigation controllers

• Optionally provides a secondary view to display search results
UISearchController

// without secondary search results controller
let searchController = UISearchController()
searchController.searchResultsUpdater = self
searchController.obscuresBackgroundDuringPresentation = false
navigationItem.searchController = searchController

// with secondary search results controller
let searchController = UISearchController(searchResultsController: contactSearchResultsViewController)
searchController.searchResultsUpdater = self
navigationItem.searchController = searchController

// UISearchResultsUpdating conformance
func updateSearchResults(for searchController: UISearchController) {
    if let searchString = searchController.searchBar.text, searchString.isEmpty == false {
        filteredItems = items.filter { (item) -> Bool in
            item.localizedDescriptionInsensitiveContains(searchString)
        }
    } else {
        filteredItems = items
    }
    // reload view
}
Handling Data Changes

• reloadData()

• performBatchUpdates()
## Diffable Data Sources

<table>
<thead>
<tr>
<th>Index</th>
<th>Original</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cat</td>
<td>Monkey</td>
</tr>
<tr>
<td>2</td>
<td>Dog</td>
<td>Giraffe</td>
</tr>
<tr>
<td>3</td>
<td>Elephant</td>
<td>Horse</td>
</tr>
<tr>
<td>4</td>
<td>Fish</td>
<td>Fish</td>
</tr>
<tr>
<td>5</td>
<td>Horse</td>
<td>Sugar Glider</td>
</tr>
<tr>
<td>6</td>
<td>Giraffe</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Monkey</td>
<td></td>
</tr>
</tbody>
</table>

### Events:

1. Cat removed
2. Dog removed
3. Elephant removed
4. Fish no change
5. Horse moved to 3
6. Giraffe moved to 2
7. Monkey moved to 1
8. Sugar Glider inserted at 5
Handling Data Changes

• reloadData()

• performBatchUpdates()

• Diffable data source

• UICollectionViewDiffableDataSource<SectionIdentifierType, ItemIdentifierType>

• NSDiffableDataSourceSnapshot<SectionIdentifierType, ItemIdentifierType>
Diffable Data Sources

- Container used by UICollectionViewDiffableDataSource
- Allows insertion, removal, and moving of items and sections
- Identifier-based rather than IndexPath-based
- Generic type with two type requirements
  - SectionIdentifierType
  - ItemIdentifierType
- Identifiers must conform to Hashable
// setup data source and apply originating snapshot
let dataSource = UICollectionViewDiffableDataSource<String, String>(collectionView: collectionView) { (collectionView, indexPath, item) -> UICollectionViewCell? in
    // cell configuration code
}
var original = NSDiffableDataSourceSnapshot<String, String>()
original.appendSections(["Main"栠]
original.appendItems(["Cat", "Dog", "Elephant", "Fish", "Horse", "Giraffe", "Monkey"], toSection: "Main")

datasource.apply(original, animatingDifferences: true, completion: nil)

// events occurring over time and apply
var updated = dataSource.snapshot()
updated.deleteItems(["Cat", "Dog", "Elephant"栠]
dataSource.apply(updated, animatingDifferences: true, completion: nil)

updated.moveItem("Horse", beforeItem: "Fish")
dataSource.apply(updated, animatingDifferences: true, completion: nil)

updated.moveItem("Giraffe", beforeItem: "Horse")
dataSource.apply(updated, animatingDifferences: true, completion: nil)

updated.moveItem("Monkey", beforeItem: "Giraffe")
dataSource.apply(updated, animatingDifferences: true, completion: nil)
// setup data source and apply originating snapshot
let dataSource = UICollectionViewDiffableDataSource
<String, String>(collectionView: collectionView) {
    collectionView, indexPath, item in
        // cell configuration code
}

var original = NSDiffableDataSourceSnapshot
<String, String>()
original.appendSections(["Main"])
original.appendItems(["Cat", "Dog", "Elephant", "Fish", "Horse", "Giraffe", "Monkey"], toSection: "Main")

dataSource.apply(original, animatingDifferences: true, completion: nil)

// events occurring all at once
var updated = NSDiffableDataSourceSnapshot
<String, String>()
updated.appendSections(["Main"])
updated.appendItems(["Monkey", "Giraffe", "Horse", "Fish", "Sugar Glider"], toSection: "Main")

dataSource.apply(updated, animatingDifferences: true, completion: nil)
Compositional Layout
UICollectionViewCompositionalLayout

- A subclass of UICollectionViewLayout
- Compose layouts by grouping smaller layouts together
- Layout groups are line based
- Compose complex layouts
UICollectionViewCompositionalLayout

- Three core components
  - Item (NSCollectionLayoutItem)
  - Group (NSCollectionLayoutGroup)
  - Section (NSCollectionLayoutSection)
NSCollectionLayoutSize

- Items and groups are sized using NSCollectionLayoutSize
- Defined with a widthDimension and heightDimension
- NSCollectionLayoutDimension
  - Four ways to define

```swift
class NSCollectionLayoutDimension {
    class func fractionalWidth(_ fractionalWidth: CGFloat) -> Self
    class func fractionalHeight(_ fractionalHeight: CGFloat) -> Self
    class func absolute(_ absoluteDimension: CGFloat) -> Self
    class func estimated(_ estimatedDimension: CGFloat) -> Self
}

let size = NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.25),
                                  heightDimension: .fractionalWidth(0.25))
```
NSCollectionLayoutItem

- Item represents a cell or supplementary view
- Defined with a size

```swift
let size = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                 heightDimension: .absolute(44.0))

let item = NSCollectionLayoutItem(layoutSize: size)
```
NSCollectionLayoutGroup

- Contains one or many items
- Defined as horizontal, vertical, or custom
- Also defined with a size
- Subclass of NSCollectionLayoutItem

```swift
let size = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                  heightDimension: .absolute(44.0))

let item = NSCollectionLayoutItem(layoutSize: size)

let group = NSCollectionLayoutGroup.horizontal(
             layoutSize: size, subitems: [item])
```
NSCollectionLayoutSection

- Contains single type of group layout item
- Configurable to scroll orthogonally to the collection view

```swift
let size = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                 heightDimension: .absolute(44.0))

let item = NSCollectionLayoutItem(layoutSize: size)

let group = NSCollectionLayoutGroup.horizontal(
            layoutSize: size, subitems: [item])

let section = NSCollectionLayoutSection(group: group)
section.orthogonalScrollingBehavior = .continuous
```
Orthogonal Scrolling

- **NSCollectionLayoutSection.orthogonalScrollingBehavior**
- **Set to a value of UICollectionViewLayoutSectionOrthogonalScrollingBehavior**

<table>
<thead>
<tr>
<th>Case</th>
<th>Scrolling Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>.none</td>
<td>The section does not allow users to scroll its content orthogonally.</td>
</tr>
<tr>
<td>.continuous</td>
<td>The section allows users to scroll its content orthogonally with continuous scrolling.</td>
</tr>
<tr>
<td>.continuousGroupLeadingBoundary</td>
<td>The section allows users to scroll its content orthogonally, coming to a natural stop at the leading boundary of the visible group.</td>
</tr>
<tr>
<td>.paging</td>
<td>The section allows users to page its content orthogonally.</td>
</tr>
<tr>
<td>.groupPaging</td>
<td>The section allows users to page its content orthogonally one group at a time.</td>
</tr>
<tr>
<td>.groupPagingCentered</td>
<td>The section allows users to page its content orthogonally one group at a time, centering each group.</td>
</tr>
</tbody>
</table>

- The section lays out its content along the main axis of its layout, defined by the layout's `scrollDirection` property
Compositional Layout

- Layout is defined with a section

```swift
let size = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                  heightDimension: .absolute(44.0))

let item = NSCollectionLayoutItem(layoutSize: size)

let group = NSCollectionLayoutGroup.horizontal(
              layoutSize: size, subitems: [item])

let section = NSCollectionLayoutSection(group: group)
section.orthogonalScrollingBehavior = .continuous

let layout = UICollectionViewCompositionalLayout(section: section)
```
Advanced Compositional Layout

- Layout can be defined with a `sectionProvider` closure.
- Closure can return distinct `NSCollectionLayoutSection` instances.

```swift
let layout = UICollectionViewCompositionalLayout {
    (section, layoutEnvironment) -> NSCollectionLayoutSection? in
    switch section {
    case 0:
        // return NSCollectionLayoutSection instance for section index 0
    case 1:
        // return NSCollectionLayoutSection instance for section index 1
    case 2:
        // return NSCollectionLayoutSection instance for section index 2
    default:
        return nil
    }
}
```
Multiple Layouts

- Layouts are UICollectionViewCompositionalLayout classes
- Can define variables with different layouts
- Call setCollectionViewLayout(_:animated:) to transition
- Transitions can be animated

```swift
func generateNewLayout() -> UICollectionViewCompositionalLayout {
    // create new layout and return it
}

collectionView.setCollectionViewLayout(generateNewLayout(), animated: true)
```
Compositional Layout Supplementary Views

• Attach to NSCollectionLayoutItem

• Additional view to add adornment such as a badge or selection indicator

• Positioned using NSCollectionLayoutAnchor

• Anchored to items or groups using NSDirectionalRectEdge
  • top
  • leading
  • trailing
  • bottom
Compositional Layout Supplementary Views

- Sections support `NSCollectionLayoutBoundarySupplementaryItem`
- Anchored to section via the `boundarySupplementaryItems` property

```swift
let section = NSCollectionLayoutSection(group: group)

let headerSize = NSCollectionLayoutSize(
    widthDimension: .fractionalWidth(1.0),
    heightDimension: .absolute(28))

let headerItem = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: headerSize,
    elementKind: "Header", alignment: .topLeading)

section.boundarySupplementaryItems = [headerItem]
```
Compositional Layout Supplementary Views

• Sections support NSCollectionLayoutDecorationItem

• Anchored to section via the decorationItems property

• Used to add a background to a section

```swift
let section = NSCollectionLayoutSection(group: group)

let background = NSCollectionLayoutDecorationItem.background(elementKind: "Background")
section.decorationItems = [background]

// once the layout is defined
layout.register(BackgroundSupplementaryView.self, forDecorationViewOfKind: "Background")
```