

**ỦY BAN NHÂN DÂN TP. HỒ CHÍ MINH
TRƯỜNG CAO ĐẲNG CÔNG NGHỆ THỦ ĐỨC
KHOA CÔNG NGHỆ THÔNG TIN**

GIÁO TRÌNH

HỌC PHẦN: LẬP TRÌNH DI ĐỘNG TRÊN iOS

NGÀNH/NGHỀ: CÔNG NGHỆ THÔNG TIN

TRÌNH ĐỘ: CAO ĐẲNG

*Ban hành kèm theo Quyết định số:... .../QĐ-CNTĐ-CN ngày.... tháng.... năm...
của.....*

TP. Hồ Chí Minh, năm 2020

TUYÊN BỐ BẢN QUYỀN

Tài liệu này thuộc loại sách giáo trình nên các nguồn thông tin có thể được phép dùng nguyên bản hoặc trích dùng cho các mục đích về đào tạo và tham khảo.

Mọi mục đích khác mang tính lèch lạc hoặc sử dụng với mục đích kinh doanh thiêu lành mạnh sẽ bị nghiêm cấm.

LỜI GIỚI THIỆU

Giáo trình *Lập trình di động trên iOS* được viết cho học phần chuyên ngành bắt buộc, ngành Công nghệ Thông tin, trường Cao đẳng Công nghệ Thủ Đức. Giáo trình được viết theo hướng tích hợp và khuyến khích tính chủ động tích cực từ người học. Khác với lập trình Android, sinh viên đã có kiến thức về Java trước khi học, môn học này với chỉ 75 tiết, nhưng giáo trình đã vừa tích hợp việc giảng dạy cho sinh viên ngôn ngữ lập trình mới – Swift, vừa giảng dạy cho sinh viên bộ công cụ Xcode để có thể phát triển các ứng dụng iOS trên nền ngôn ngữ Swift, đồng thời cũng vừa hoàn thiện, cải tiến các kỹ năng về phân tích, thiết kế phần mềm, tư duy lập trình, phân tích thiết kế cơ sở dữ liệu... thông qua việc từng bước phân tích, lập trình và hoàn thiện hai ứng dụng thực tiễn thuần nhất trong toàn bộ giáo trình. Điều này sẽ giúp sinh viên có những kiến thức, kỹ năng gần hơn với nhu cầu của doanh nghiệp và giúp các em có hứng thú hơn trong suốt quá trình học tập của mình.

Giáo trình viết ngắn gọn, nhưng đầy đủ các kiến thức, kỹ năng cần thiết về phát triển các ứng dụng trên iOS. Giáo trình được chia thành 4 chương cơ bản lần lượt trình bày về ngôn ngữ Swift, Xcode và phát triển ứng dụng iOS với Xcode, phân tích, thiết kế và làm việc với cơ sở dữ liệu cũng như bản đồ trực tuyến trên ứng dụng iOS. Mặc dù giáo trình được tách biệt thành 4 chương, nhưng trong quá trình giảng dạy, giảng viên cần linh động trong việc tích hợp và giảng dạy đan xen kiến thức và kỹ năng của ngôn ngữ mới Swift (Chương 1) với cách phân tích, thiết kế và xây dựng các ứng dụng cơ bản trên iOS dùng Xcode (Chương 2) thì mới có thể đảm bảo thời lượng của môn học. Còn chương 3 và chương 4 giảng viên giảng dạy cần tích hợp quá trình phân tích, thiết kế và triển khai từng bước của ứng dụng thực tiễn cũng như yêu cầu sinh viên thực hiện theo hệ thống các bài tập đã đề ra của mỗi phần thì mới đạt yêu cầu.

Thành phố Hồ Chí Minh, ngày 15 tháng 01 năm 2021
Giảng viên biên soạn

Tiêu Kim Cường

MỤC LỤC

	TRANG
DANH MỤC CÁC TỪ VIẾT TẮT	VI
DANH MỤC BẢNG BIỂU VÀ HÌNH VẼ	VII
CHƯƠNG 1. TỔNG QUAN VỀ PHÁT TRIỂN ỨNG DỤNG TRÊN IOS VÀ NGÔN NGỮ SWIFT	2
1.1 Tổng quan về phát triển ứng dụng trên iOS	2
1.1.1 IOS là gì? Kiến trúc bên trong iOS?	2
1.1.2 Mô hình MVC trong lập trình iOS.....	3
1.1.3 Phát triển ứng dụng trên iOS	4
1.1.4 Viết ứng dụng đầu tiên trên iOS	4
1.2 Ngôn ngữ lập trình Swift	8
1.2.1 Cơ bản về ngôn ngữ lập trình Swift.....	8
1.2.2 Kiểu dữ liệu, biến, hằng	9
1.2.3 Biến kiểu Optional	10
1.2.4 Biến Tuples	12
1.2.5 Chuỗi thoát trong Swift.....	13
1.2.6 Các loại toán tử trong ngôn ngữ Swift.....	13
1.2.7 Rẽ nhánh và vòng lặp trong Swift	15
1.2.8 Xử lý chuỗi và ký tự trong Swift	20
1.2.9 Mảng trong Swift	21
1.2.10 Tập hợp trong Swift	23
1.2.11 Làm việc với kiểu Dictionaries	25
1.2.12 Lệnh guard	27
1.2.13 Hàm trong Swift.....	27
1.2.14 Định nghĩa và sử dụng Cấu trúc, Lớp đối tượng trong Swift	33
1.2.15 Ép kiểu	36
1.2.16 Protocol và cơ chế Delegate trong Swift	38
1.3 Câu hỏi và bài tập chương 1	41
CHƯƠNG 2. THIẾT KẾ GIAO DIỆN VÀ XỬ LÝ SỰ KIỆN TRÊN IOS	46
2.1 Thiết kế giao diện với Storyboard	46
2.1.1 Các thành phần chính trong iOS Project.....	46
2.1.2 Màn hình chờ LaunchScreen.storyboard	49
2.1.3 Màn hình thiết kế giao diện Main.storyboard.....	49
2.1.4 Case Study: Tạo ứng dụng Calculate.....	53

2.2 Xử lý sự kiện trên iOS	56
2.2.1 Kết nối các đối tượng với code	56
2.2.2 Cách viết hàm trong iOS với ngôn ngữ Swift.....	58
2.2.3 Hoàn thiện ứng dụng Calculate giai đoạn 1	59
2.3 Tổ chức code theo mô hình MVC	62
2.3.1 Phân tích ứng dụng theo mô hình MVC	62
2.3.2 Xây dựng ứng dụng theo mô hình MVC	63
2.3.3 Mở rộng và hoàn thiện ứng dụng Calculate.....	64
2.4 Autolayout trong iOS.....	68
2.4.1 Vấn đề giao diện trong ứng dụng Calculate.....	68
2.4.2 Cải tiến giao diện cho ứng dụng Calculate	68
2.5 Case Study: Thiết kế ứng dụng Quản lý món ăn	71
2.5.1 Luyện tập thiết kế giao diện cơ bản và autolayout trong iOS.....	71
2.5.2 Xử lý sự kiện với các Component cơ bản.....	73
2.5.3 Làm việc với UITapGestureRecognizer	77
2.5.4 Xây dựng Control mới cho ứng dụng	79
2.5.5 Thêm thuộc tính vào Attributes Inspector	84
2.5.6 Table view.....	87
2.5.7 Navigation và truyền tham số giữa các màn hình ứng dụng.....	92
2.6 Câu hỏi và bài tập chương 2	103
CHƯƠNG 3. LÀM VIỆC VỚI CƠ SỞ DỮ LIỆU.....	105
3.1 Thiết kế Data model cho ứng dụng.....	105
3.1.1 Phân tích ứng dụng Quản lý món ăn.....	105
3.1.2 Thiết kế Datamodel cho ứng dụng.....	105
3.2 Kiểm thử tính đúng đắn của Data model	106
3.2.1 Xây dựng các test cases kiểm thử tính đúng đắn cho Datamodel của ứng dụng	106
3.2.2 Kiểm thử và điều chỉnh.....	107
3.3 Một số dạng lưu trữ dữ liệu lâu dài trên ứng dụng iOS	108
3.3.1 Core Data	108
3.3.2 SQLite	108
3.3.3 Lưu trữ trên mạng	108
3.4 Cơ sở dữ liệu SQLite với các ứng dụng iOS	108
3.4.1 Cài đặt thư viện SQLite và Framework FMDB.....	108
3.4.2 Thiết kế tầng truy xuất dữ liệu DAL.....	110

3.5 Xây dựng tầng truy xuất dữ liệu cho ứng dụng iOS	111
3.5.1 Thiết kế các chức năng cơ bản: Đóng, mở, tạo bảng	111
3.5.2 Thiết kế các API cho tầng trên	113
a. Đọc dữ liệu từ cơ sở dữ liệu	113
b. Ghi dữ liệu vào cơ sở dữ liệu	114
c. Cập nhật dữ liệu vào cơ sở dữ liệu	114
d. Xoá dữ liệu từ cơ sở dữ liệu	115
e. Tìm kiếm dữ liệu từ cơ sở dữ liệu	115
3.6 Sử dụng tầng DAL cho ứng dụng iOS	115
3.7 Câu hỏi và bài tập chương 3	117
CHƯƠNG 4. BẢN ĐỒ TRỰC TUYẾN TRÊN IOS.....	118
4.1 Một số dạng bản đồ trực tuyến thông dụng	118
4.1.1 Map Kit	118
4.1.2 Google Map	119
4.2 Sử dụng bản đồ trực tuyến trong các ứng dụng iOS	119
4.2.1 Tích hợp MapKit vào ứng dụng	119
4.2.2 Tạo mới bản đồ trong ứng dụng iOS	119
4.2.3 Đánh dấu trên bản đồ dùng Annotation	121
4.2.4 Lấy vị trí hiện tại của người dùng trên bản đồ	122
4.2.5 Tương tác với các Annotation trên bản đồ	123
4.2.6 Xác định vị trí và di chuyển trên bản đồ	123
4.2.7 Đánh dấu vị trí bằng sự kiện LongPressGesture	124
4.2.8 Chỉ đường	124
4.3 Câu hỏi và bài tập chương 4	127
TÀI LIỆU THAM KHẢO	129

DANH MỤC CÁC TỪ VIẾT TẮT

STT	Từ viết tắt	Cụm từ gốc	Ghi chú
1	iOS	iPhone Operation System	
2	MVC	Models – Views – Controllers	
3	IB	Interface Builder	
4	FMDB	Flying Meat Database	

DANH MỤC BẢNG BIẾU VÀ HÌNH VẼ

Hình 1.1.1 Kiến trúc của iOS	3
Hình 1.1.4.1 màn hình khởi động Xcode.....	5
Hình 1.1.4.2 Giao diện lựa chọn một Template cho ứng dụng iOS	5
Hình 1.1.4.3 Cấu hình cho một Project iOS	6
Hình 1.1.4.4 Màn hình khởi tạo của Project iOS mới	7
Hình 1.1.4.5 Các vùng cơ bản trong bộ công cụ phát triển ứng dụng trên iOS.....	7
Bảng 1.2.1 Một số cú pháp cơ bản của ngôn ngữ Swift	9
Bảng 1.2.2 Một số kiểu dữ liệu thông dụng trong Swift.....	10
Bảng 1.2.5. Chuỗi thoát trong Swift	13
Hình 1.2.7.1 Hoạt động của câu lệnh rẽ nhánh loại 1	15
Hình 1.2.7.2 Hoạt động của câu lệnh rẽ nhánh loại 2.....	15
Hình 1.2.7.3 Hoạt động vòng lặp while.....	18
Hình 1.2.7.4 Hoạt động vòng lặp repeat ... while	19
Bảng 1.2.8.1 Một số thao tác xử lý chuỗi thông dụng.....	21
Hình 2.1.1.1 Các thành phần chính của iOS Project	46
Hình 2.1.1.2 Màn hình Debug chương trình trong iOS	48
Hình 2.1.1.3 Cấu trúc iOS Project trong Xcode (trái) và trong thư mục dự án (phải) .48	48
Hình 2.1.2.1 Cấu hình cho màn hình LaunchScreen trong ứng dụng iOS	49
Hình 2.1.3.1 Màn hình thiết kế giao diện Main.storyboard.....	50
Hình 2.1.3.2 Thư viện các đối tượng cho thiết kế giao diện (Object Library)	51
Hình 2.1.3.3 Kéo – thả đối tượng từ thư viện vào màn hình thiết kế giao diện	51
Hình 2.1.3.4 Bảng thuộc tính đối tượng trong thiết kế giao diện iOS	52
Hình 2.1.3.5 Ví dụ thay đổi thuộc tính đối tượng trực tiếp trên màn hình thiết kế	53
Hình 2.1.4.1 Tạo ứng dụng iOS Calculator2020 (a).....	53
Hình 2.1.4.2 Tạo ứng dụng iOS Calculator2020 (b)	54
Hình 2.1.4.3 Tạo ứng dụng iOS Calculator2020 (c).....	54
Hình 2.1.4.4 Gom nhóm các file hệ thống vào một nhóm để quản lý	55
Hình 2.1.4.5 Kết quả gom nhóm các file hệ thống trong System files	55
Hình 2.2.1.1 Màn hình liên kết code trong lập trình iOS	56
Hình 2.2.1.2 Thiết lập tham số cho việc liên kết code với đối tượng trong layout	57
Hình 2.2.1.3 Kết quả liên kết code dạng hành vi (Action) của đối tượng	57
Hình 2.2.2.1 Kết quả điều chỉnh giao diện giai đoạn 1	59
Hình 2.2.2.2 Kết quả điều chỉnh chương trình	60
Hình 2.2.2.3 Bổ sung phím chức năng cho máy tính	61
Hình 2.2.2.4 Lấy các ký tự đặc biệt trong Xcode	61
Hình 2.3.3.1 Giao diện chức năng bổ sung.....	66
Hình 2.4.1.1 Vấn đề thay đổi hướng nhìn của các giao diện iOS.....	68
Hình 2.4.2.1 Nhúng các đối tượng trong Stack View.....	69
Hình 2.4.2.2 Kết quả sau khi nhúng các đối tượng trong Stack View	69
Hình 2.4.2.3 Thêm ràng buộc cho Autolayout.....	70

Hình 2.4.2.4 Autolayout với các chiều khác nhau của màn hình iPhone	71
Hình 2.5.1.1 Giao diện màn hình chi tiết của một món ăn	72
Hình 2.5.1.2 Đưa bộ ảnh default vào ứng dụng iOS	72
Hình 2.5.2.1 Các trạng thái và cách chuyển trạng thái của một Controller.....	73
Hình 2.5.2.2 Kết nối đối tượng Tap Gesture Recognizer với ImageView	77
Hình 2.5.4.1 Tạo lớp RatingControl mới.....	80
Hình 2.5.4.2 Layout của RatingControl	81
Hình 2.5.4.3 Ba bộ ảnh cho đối tượng RatingControl.....	83
Hình 2.5.4.4 Thuộc tính mới trong Attributes Inspector	85
Hình 2.5.6.1 Thiết kế giao diện một phần tử trong danh sách các món ăn	87
Hình 2.5.6.1 Tạo màn hình giao diện mới với Table View Controller	88
Hình 2.5.6.2 Lựa chọn Prototype Cell để thiết kế và cấu hình.....	89
Hình 2.5.6.3 Thiết kế cho Prototype Cell	90
Hình 2.5.7.1 Sau khi nhúng Table View Controller vào Navigation Controller.....	93
Hình 2.5.7.2 Màn hình MealDetail sau khi được đưa vào Navigation Stack.....	95
Hình 2.5.7.3 Kết quả sau khi nhúng MealDetail vào Navigation Controller mới	96
Hình 2.5.7.4 Tạo Unwind Segue cho nút Save và kết nối với Unwind Action.....	99
Hình 2.5.7.5 Hiện thực hoá chức năng cập nhật một món ăn đã có	101
Hình 3.2.1.1 Mở file để viết các Unit Test trong iOS	106
Hình 3.2.2.1 Chỉ dừng lại khi mọi test cases đều pass hết	107
Hình 3.4.1.1 Cài đặt thư viện SQLite cho ứng dụng iOS	109
Hình 3.4.1.2 Download code source của FMDB.....	109
Hình 3.4.1.3 Tuỳ chọn tích hợp thư viện FMDB vào Project	110
Hình 3.4.1.4 Tạo cầu nối thư viện Objective-C với ngôn ngữ Swift.....	110
Hình 4.2.2.1 Tạo màn hình MapViewController.....	120
Hình 4.2.3.1 Đánh dấu trên bản đồ	121
Hình 4.2.4.1 Cấp quyền cho việc truy xuất vị trí hiện tại trên bản đồ.....	122
Hình 4.2.4.2 Yêu cầu cấp quyền truy xuất	123
Hình 4.2.8.1 Màn hình MapView với chức năng chỉ đường	126
Hình 4.3.1 Cải tiến chức năng 4.2.5	127

GIÁO TRÌNH HỌC PHẦN

Tên học phần: LẬP TRÌNH DI ĐỘNG TRÊN iOS

Mã học phần: CNC107440

Vị trí, tính chất, ý nghĩa và vai trò của học phần:

- Vị trí: Học phần chuyên ngành bắt buộc.
- Tính chất: Đây là học phần bắt buộc ngành Công nghệ thông tin, cung cấp cho sinh viên một cái nhìn tổng quát về phương pháp thiết kế ứng dụng trên thiết bị di động thông qua ngôn ngữ Swift cũng như các công cụ lập trình để phát triển ứng dụng cho hệ điều hành iOS. Qua đó, Sinh viên vận dụng được kỹ năng về thiết kế phần mềm để xây dựng các ứng dụng chạy trên iOS. Thông qua các hoạt động học tập, giúp Sinh viên rèn luyện tư duy giải quyết vấn đề và tuân thủ các quy định trong khi tham gia vào các dự án vừa và nhỏ.

Mục tiêu của học phần:

- Về kiến thức:
 - + Hiểu ngôn ngữ Swift và Thiết kế được các giao diện cơ bản cho các ứng dụng trên iOS;
 - + Vận dụng được mô hình MVC vào phát triển ứng dụng trên iOS
- Về kỹ năng:
 - + Thiết kế và xây dựng được cơ sở dữ liệu cho các ứng dụng trên iOS;
 - + Xây dựng được các ứng dụng vừa và nhỏ sử dụng Webservice và các dịch vụ cơ bản (Map và Location);
 - + Hiện thực hóa các ứng dụng vừa và nhỏ trên iOS;
 - + Xây dựng được bản Phân tích, thiết kế cho một sự án phần mềm vừa và nhỏ;
 - + Sử dụng các công cụ quản lý khi triển khai Project trong nhóm.
- Về năng lực tự chủ và trách nhiệm:
 - + Luôn chủ động tìm hiểu vấn đề khi thực hiện các nhiệm vụ được giao;
 - + Luôn tuân thủ đầy đủ các quy định của lớp học.

CHƯƠNG 1. TỔNG QUAN VỀ PHÁT TRIỂN ỨNG DỤNG TRÊN IOS VÀ NGÔN NGỮ SWIFT

Mục tiêu: Hiểu ngôn ngữ Swift và một số khái niệm cơ bản trong lập trình trên iOS.

Mô tả nội dung: Trình bày tổng quan về phát triển ứng dụng di động trên iOS và đi sâu vào tìm hiểu các thành phần của ngôn ngữ lập trình Swift.

1.1 Tổng quan về phát triển ứng dụng trên iOS

1.1.1 iOS là gì? Kiến trúc bên trong iOS?

iOS hay iPhone OS có thể hiểu là hệ điều hành cho các thiết bị di động (iPhone, iPad...) được phát triển bởi Apple. Ban đầu nó được thiết kế cho iPhone, nhưng hiện nay đã hỗ trợ cho các dạng thiết bị khác của Apple như iPod touch, iPad, Apple TV... iOS ra đời và ngày càng phát triển qua nhiều thế hệ: iOS 1 ra đời năm 2007 cùng với sự xuất hiện của iPhone. Đến năm 2008, iOS 2 ra đời bổ sung thêm tính năng App Store. Năm 2009 tính năng iPhone Enhancement được bổ sung cùng với sự xuất hiện của iOS 3. Năm 2010 đánh dấu một bước đột phá của Apple khi công bố iOS 4 với MultiTasking, Retina và FaceTime... và cho đến hiện nay (2020) chúng ta đang được sử dụng thế hệ iOS 14 với 8 tính năng bổ sung: Cho phép sử dụng một trình duyệt khác Safari làm trình duyệt mặc định; App Library mới trên iOS 14 chứa các Widgets màn hình chính và ứng dụng vào trong một thư mục và thay thế cho giao diện trang Ứng dụng trước đó với chế độ xem đơn giản và dễ điều hướng; Giao diện cuộc gọi được cải tiến được thu gọn lại dưới dạng biểu tượng giống như các thông báo khác; Màn hình home có thể thêm các Widgets với kích thước tùy ý trực tiếp ngay trong Home Screen; Thay đổi cách xuất hiện của trợ lý ảo Siri, thay vì chiếm cả màn hình như trước, Siri sẽ xuất hiện phía dưới màn hình, còn câu trả lời xuất hiện phía trên; Bổ sung tính năng gõ lên lưng sau của iPhone sẽ cho phép chụp ảnh màn hình, tăng giảm âm lượng (người dùng tự thiết lập cấu hình); Bổ sung tính năng Hiệu ứng âm thanh không gian trên AirPod Pro; Thu hồi lại tin nhắn đã gửi trên Message và thêm tính năng Picture in Picture cho ứng dụng YouTube trên iPhone.

Trên phương diện lập trình viên thì có thể hiểu iOS như một Sofware Stack bao gồm 4 tầng (Hình 1.1.1). Trong đó:

- Tầng Core OS: Là hạt nhân của hệ điều hành iOS dựa trên nền của Unix.

- Tầng Core Services, Media Layer, Cocoa Touch: Chứa các thư viện và dịch vụ phục vụ cho việc phát triển các ứng dụng chạy trên iOS. Tầng Core Services cung cấp các thư viện và dịch vụ để phát triển ứng dụng như dịch vụ file, tổ chức cơ sở dữ liệu SQLite, dịch vụ mạng,... Tầng Media Layer cung cấp các thư viện và dịch vụ cho phát triển ứng dụng như Audio, Video, làm việc với các file hình ảnh, PDF, ... Tầng Cocoa Touch cung cấp các thư viện còn lại cho phát triển ứng dụng trên iOS như xử lý Multi-Touch, MapKit, Các loại View và Controls, Camera, Image Picker, ...



Hình 1.1.1 Kiến trúc của iOS

1.1.2 Mô hình MVC trong lập trình iOS

MVC là viết tắt của **Model**, **View** và **Controller** trong thiết kế phần mềm (là một Software Design Pattern) giúp các nhà phát triển phần mềm tổ chức code sáng sủa, dễ hiệu chỉnh, sửa chữa và dễ bảo trì.

Với mô hình MVC, mỗi ứng dụng được tổ chức về mặt logic thành 3 nhóm đối tượng chính tách biệt: Nhóm Views, Nhóm Models và nhóm Controllers. Nhóm Views đảm trách mọi vấn đề liên quan đến xây dựng giao diện ứng dụng và mọi tương tác với người dùng. Nhóm Models đảm trách mọi vấn đề về tổ chức dữ liệu cũng như các hoạt động chuyên môn của ứng dụng đó. Nhóm Controllers giữ vai trò trung gian điều phối giữa những yêu cầu của người dùng (trên Views) và các hoạt động mang tính chuyên môn nghiệp vụ của ứng dụng. Ví dụ về ứng dụng **Máy tính bỏ túi**, với những lập trình viên không có kinh nghiệm và không biết tổ chức code theo mô hình MVC thì anh ta hoàn toàn có thể chỉ thiết kế giao diện và xây dựng một lớp duy nhất vừa tiếp nhận yêu cầu từ người sử dụng vừa dựa trên yêu cầu đó tiến hành tính toán và gửi trả lại kết quả cho

người dùng. Tuy nhiên, việc điều chỉnh, bảo trì sẽ gặp vô vàn khó khăn. Ví dụ nếu thêm một tính năng mới cho máy tính chẳng hạn sẽ có thể dẫn tới việc phải “đập đi làm lại” những gì đã viết trước đó. Nếu tổ chức ứng dụng này theo mô hình MVC thì ngoài các giao diện như đã thiết kế (vai trò chính là các Views) cần tách lớp đã xây dựng trước đó thành hai lớp: một lớp có tên **CalculatorBrain** (vai trò chính là Models trong mô hình MVC) đảm trách mọi hoạt động liên quan đến tính toán cho máy tính, nó sẽ nhận vào là các toán hạng, toán tử và trả về kết quả của phép tính; và lớp thứ 2 có tên Calculator (vai trò chính là Controllers) đảm trách điều phối trung gian giữa yêu cầu người dùng (từ Views) và các hoạt động tính toán của máy tính. Như vậy, theo mô hình này, mỗi khi người dùng gửi yêu cầu (ví dụ thực hiện phép toán $10+20$ trên giao diện) thì lớp Controller (chính là Calculator) sẽ tiếp nhận, phân tích và nhận dạng loại yêu cầu (ở đây là phép cộng giữa toán hạng 10 và toán hạng 20) và gửi đến lớp **CalculatorBrain** yêu cầu thực hiện nó đồng thời tiếp nhận kết quả trả về từ CalculatorBrain và cập nhật lên giao diện kết quả (thành phần Views) cho người dùng (Các chương sau sẽ đề cập chi tiết hơn vấn đề này).

1.1.3 Phát triển ứng dụng trên iOS

Để có thể phát triển được các ứng dụng trên iOS, người học cần đáp ứng được những yêu cầu về kiến thức, kỹ năng cũng như cần chuẩn bị trước môi trường phát triển theo những yêu cầu tối thiểu sau đây:

- Về kiến thức và kỹ năng: Thành thạo về thiết kế và phát triển ứng dụng theo tiếp cận hướng đối tượng; thành thạo về thiết kế và xây dựng cơ sở dữ liệu theo mô hình dữ liệu quan hệ.
- Hệ điều hành: Cần sử dụng hệ điều hành Mac OSX (Sử dụng các máy tính của Apple hoặc cài trên máy ảo).
- Công cụ phát triển: Cài đặt bộ công cụ Xcode (phiên bản mới nhất).

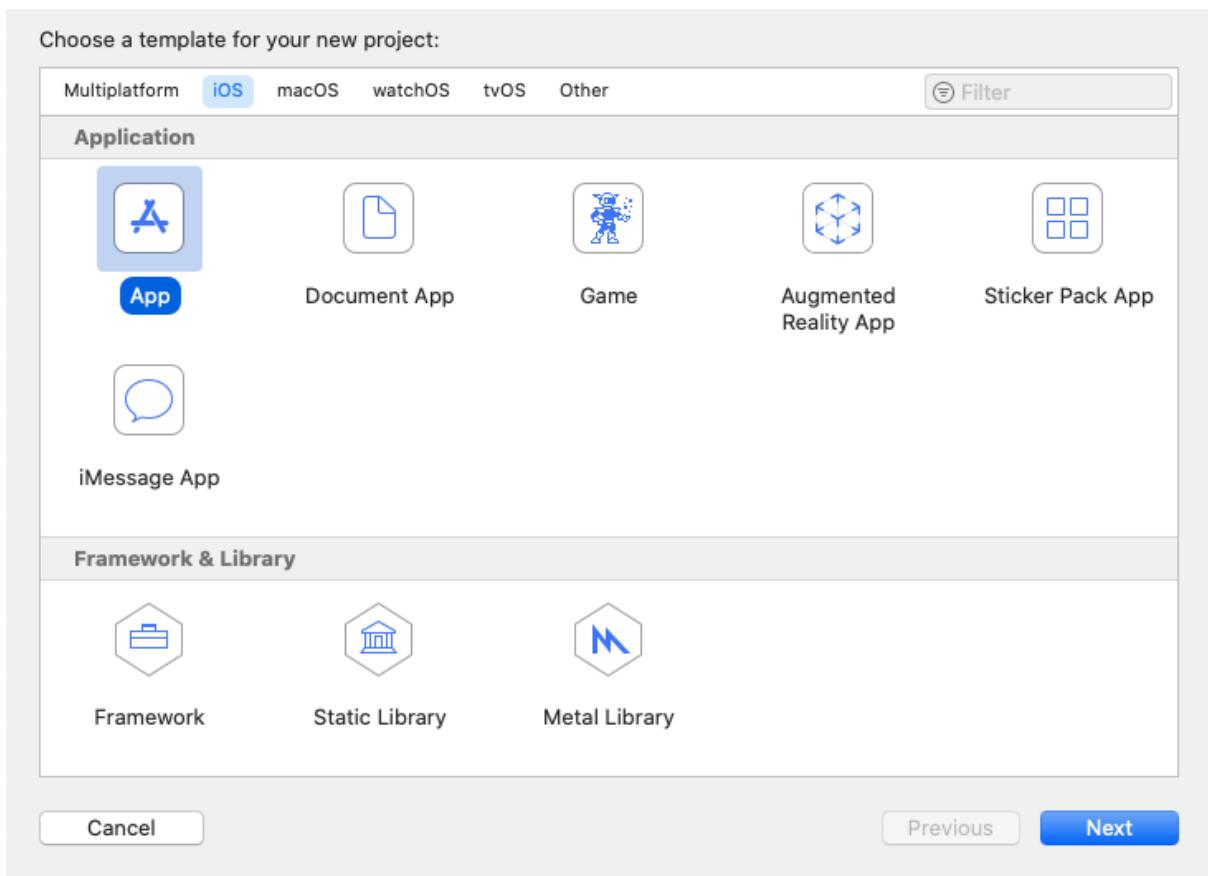
1.1.4 Viết ứng dụng đầu tiên trên iOS

Khởi động Xcode rồi thực hiện chuỗi thao tác trên thanh Menu hệ thống File => New => Project... hoặc nhập chọn Create a new Xcode Project (Hình 1.1.4.1) để tạo một Project mới cho ứng dụng iOS.

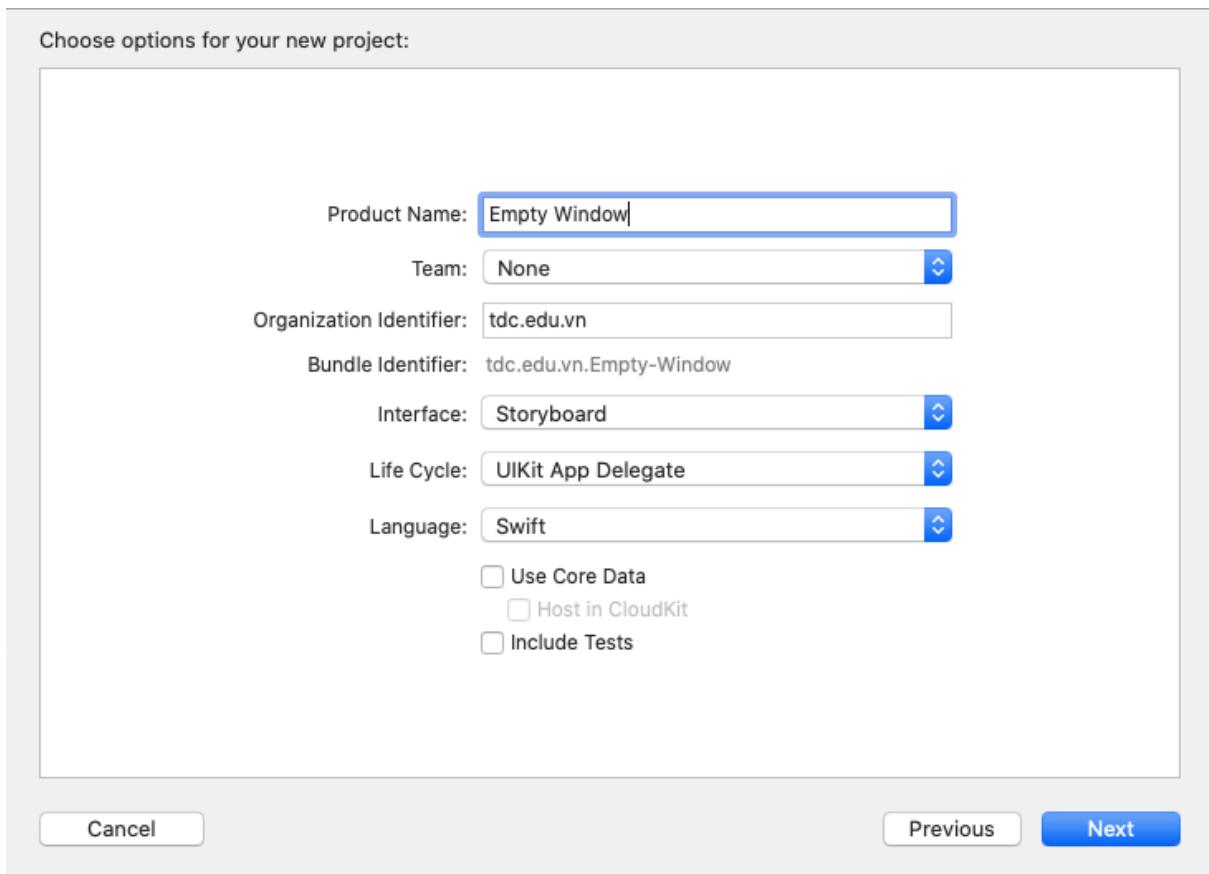


Hình 1.1.4.1 Màn hình khởi động Xcode

Tiếp theo, cần chọn một Template cho ứng dụng muốn xây dựng: Lựa chọn iOS => App và Next để tiếp tục (Hình 1.1.4.2).



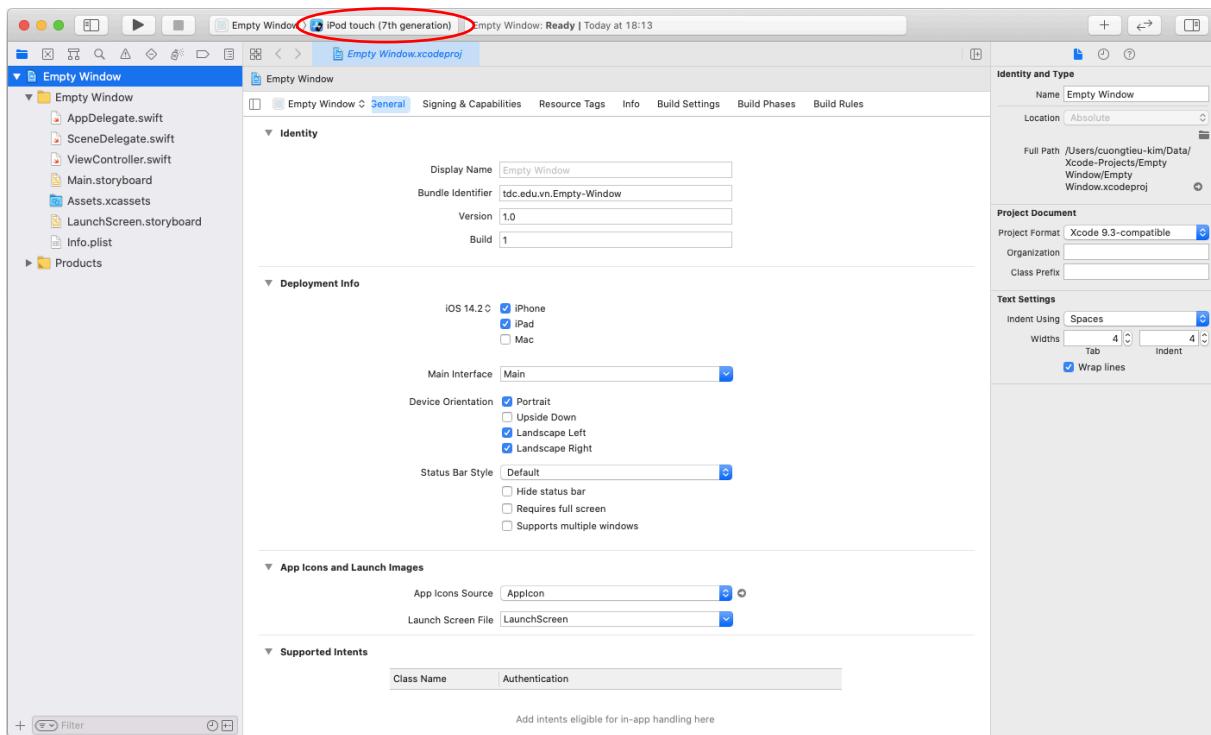
Hình 1.1.4.2 Giao diện lựa chọn một Template cho ứng dụng iOS



Hình 1.1.4.3 Cấu hình cho một Project iOS

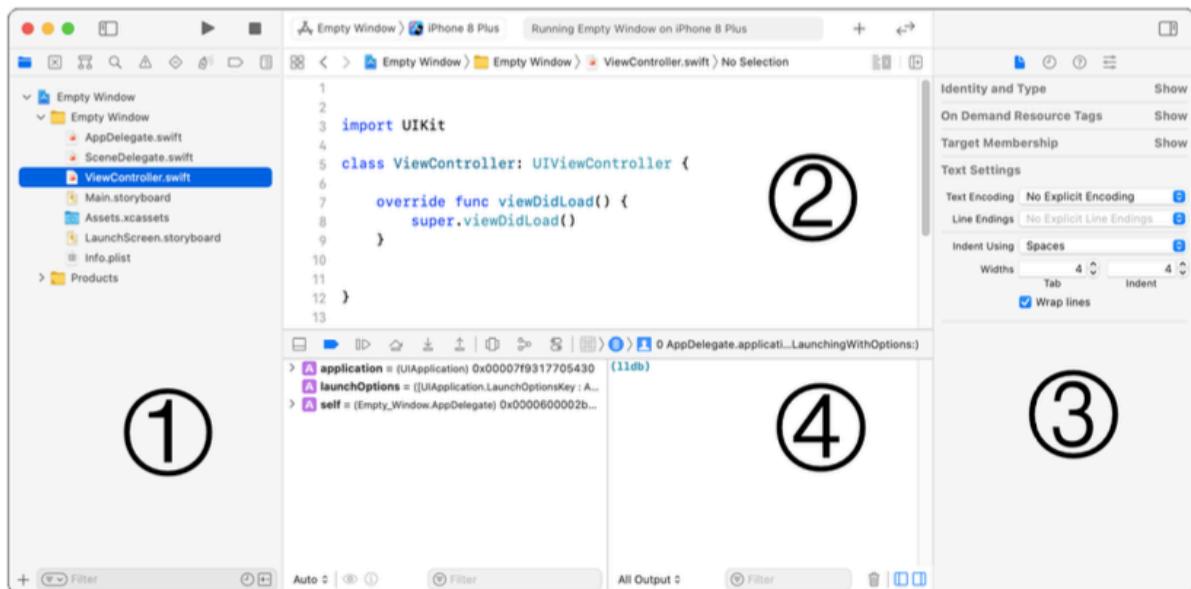
Màn hình cấu hình cho Project iOS xuất hiện (Hình 1.1.4.3). Trong mục **Product Name** hãy nhập tên ứng dụng muốn tạo (trong ví dụ là **Empty Window**), mục Team tạm thời chọn **None**, mục **Organization Identifier** gõ vào **tdc.edu.vn** (domain của TDC), mục **Interface** lựa chọn là **Storyboard** và mục **Language** chọn ngôn ngữ **Swift**. Hiện tại bỏ chọn ở hai mục **Use Core Data** và **Include Tests**. Sau khi hoàn thành chọn **Next** => Lựa chọn nơi sẽ chứa Project => Create. Xcode sẽ tạo một Project mới theo cấu hình đã lựa chọn (Hình 1.1.4.4).

Trên thanh công cụ phía trên màn hình (Vòng Elipe màu đỏ trên hình 1.1.4.4) lựa chọn loại máy ảo để chạy chương trình: Chọn **iPhone 8 Plus**. Để chạy thử ứng dụng, Click trên biểu tượng Nút “Play” trên thanh công cụ. Xcode sẽ tự động biên dịch chương trình, khởi tạo máy ảo iPhone 8 Plus, cài đặt ứng dụng trên máy ảo và chạy ứng dụng mới tạo trên máy ảo đó. Kết quả, chúng ta nhận được một màn hình trắng trên máy ảo iPhone 8 Plus. Đó chính là ứng dụng chúng ta vừa tạo và chạy trên máy ảo.



Hình 1.1.4.4 Màn hình khởi tạo của Project iOS mới

Trong Panel bên trái, nhấp chọn ViewCotroller.swift để nhìn rõ hơn giao diện của bộ công cụ phát triển ứng dụng iOS (Hình 1.1.4.5).



Hình 1.1.4.5 Các vùng cơ bản trong bộ công cụ phát triển ứng dụng trên iOS

Vùng (1) chứa bảng điều hướng dùng để xem cấu trúc Project và di chuyển qua lại giữa các thành phần trong Project. Vùng (2) là vùng soạn thảo code của chương trình. Vùng (3) là bảng thuộc tính của các đối tượng và vùng (4) dùng để xem kết quả và gỡ rối chương trình.

1.2 Ngôn ngữ lập trình Swift

1.2.1 Cơ bản về ngôn ngữ lập trình Swift

Swift là ngôn ngữ lập trình được phát triển bởi Apple từ năm 2010 nhằm thay thế cho ngôn ngữ Objective-C trong phát triển các ứng dụng iOS. Swift được thiết kế dựa trên ý tưởng từ nhiều ngôn ngữ lập trình nổi tiếng khác như Objective-C, Rust, Haskell, Ruby, Python, C# và CLU. Do vậy Swift có nhiều ưu điểm vượt trội hơn so với việc sử dụng Objective-C trong phát triển ứng dụng iOS như nhanh, đơn giản, hiệu năng xử lý tốt hơn, đặc biệt với tính năng Playground giúp lập trình viên có khả năng xem nhanh kết quả trong thời gian thực, hỗ trợ tốt cho quá trình phát triển ứng dụng... Ngoài ra, trong ứng dụng iOS viết bằng ngôn ngữ Swift vẫn hoàn toàn có thể tái sử dụng các thư viện viết bằng Objective-C trước đó. Do đó Swift sẽ là lựa chọn cho việc phát triển các ứng dụng iOS trong hiện tại và tương lai.

Dưới đây là một số cú pháp cơ bản của ngôn ngữ Swift:

STT	Cú pháp	Mô tả
1	import TenThuVien	Sử dụng thư viện có sẵn cho ứng dụng iOS. Ví dụ: import UIKit
2	// Comment	Ghi chú cho một dòng code, nội dung ghi chú sẽ bị bỏ qua khi biên dịch chương trình.
3	/* Comment */	Ghi chú trên nhiều dòng code, nội dung ghi chú sẽ bị bỏ qua khi biên dịch chương trình.
4	import UIKit var name = "Nguyen A" var greeting = "Hi" print(greeting); print(name)	Trong Swift nếu mỗi lệnh kết thúc trên 1 dòng thì không cần dùng ";" để kết thúc lệnh. Tuy nhiên, nếu có nhiều lệnh trên 1 dòng thì cần phân tách các lệnh đó bởi dấu ";".
5	Quy tắc đặt tên trong Swift: Tên trong Swift phân biệt chữ hoa, chữ thường; tên phải bắt đầu bằng chữ cái a-z hoặc A-Z, hoặc ký tự gạch dưới "_" ; tiếp theo có thể là các chữ cái chữ số và ký tự "-". Tên trong Swift không được chứa các ký tự đặc biệt như @, #, %,... (Đặt tên giống trong ngôn ngữ lập trình C).	Một số tên hợp lệ: aBC10, Abc100, move_name, abc_123... Lưu ý: Swift cũng cho phép sử dụng tên dựa trên các ký tự Unicode, nên ta hoàn toàn có thể dùng các ký tự Unicode trong bảng chữ cái của nước khác để đặt tên biến. Ví dụ khai báo sau hoàn toàn hợp lệ: var 你好 = "你好世界" print(你好)
6	Từ khoá (Keyword) trong ngôn ngữ Swift:	Một số từ khoá dùng để khai báo trong Swift:

	<p>Là những tên được dành riêng của ngôn ngữ với nhiều mục đích khác nhau (ví dụ import để sử dụng thư viện có sẵn, var để khai báo biến...). Không được đặt tên biến, tên hàm... trùng với tên từ khoá. Tuy nhiên, ngôn ngữ Swift vẫn cho phép dùng tên biến trùng với từ khoá với điều kiện đặt trước và sau nó ký tự '' (Ví dụ `import` là tên hợp lệ, còn import thì không).</p>	<p>class, _deinit, enum, extension, func, import, init, internal, let, operator, private, protocol, public, static, struct, subscript, typealias, var</p> <p>Một số từ khoá dùng trong các câu lệnh: break, case, continue, default, do, else, fallthrough, for, if, in, return, switch, where, while</p> <p>Một số từ khoá dùng trong biểu thức: as, false, is, nil, self, super, true</p> <p>Một số từ khoá dùng trong ngữ cảnh đặc biệt: associatedtype, dynamic, didSet, final, get, inout, optional, override, required</p>
7	<p>Quy tắc sử dụng dấu cách (space) trong Swift:</p> <p>Dấu cách (dấu khoảng trắng) được dùng để phân tách các từ trong một câu lệnh Swift.</p>	<p>Ví dụ: var age; thì giữa từ khoá var và biến age có thể có 1 hoặc nhiều khoảng trắng (dấu cách). Tuy nhiên, khác với các ngôn ngữ khác số lượng dấu cách (dấu khoảng trắng) ở hai bên toán tử bắt kỳ phải bằng nhau:</p> <p>Ví dụ đúng: Int a = 10</p> <p>Ví dụ sai: Int a= 10</p>

Bảng 1.2.1 Một số cú pháp cơ bản của ngôn ngữ Swift

1.2.2 Kiểu dữ liệu, biến, hằng

Trong Swift có một số kiểu dữ liệu cơ bản sau:

STT	Tên kiểu dữ liệu	Mô tả
1	Int/UInt	Dùng để khai báo các biến kiểu số nguyên có dấu/không dấu. Phạm vi biểu diễn: từ -2147483648 tới 2147483647 (Với số nguyên 32 bit có dấu) hoặc từ 0 tới 4294967295 (Với số nguyên không dấu 32 bit).
2	Float	Dùng để khai báo các biến kiểu số thực. Phạm vi biểu diễn từ 1.2E-38 tới 3.4E+38 (Với số thực kiểu 4 bytes, độ chính xác 6 số sau dấu phẩy thập phân).
3	Double	Dùng để khai báo các biến kiểu số thực có độ chính xác kép. Phạm vi biểu diễn từ 2.3E-308 tới 1.7E+308 (Với số thực kiểu 8 bytes, độ chính xác kép 15 số sau dấu phẩy thập phân).
4	Bool	Dùng để khai báo các biến kiểu logic, chỉ nhận các giá trị đúng hoặc sai (true/false).
5	String	Dùng để khai báo các biến kiểu chuỗi ký tự (ví dụ như String name = "Nguyen Van A").
6	Character	Dùng để khai báo các biến kiểu ký tự đơn.
7	Optional	Dùng để khai báo một dạng biến đặc biệt trong Swift (sẽ được làm rõ trong phần sau).
8	Tuples	Dùng để định nghĩa các biến được tạo thành từ nhiều giá trị đơn lẻ khác nhau trong Swift (sẽ được làm rõ trong phần sau).

9	Type alias	Kiểu dữ liệu bí danh. Trong ngôn ngữ Swift cho phép người dùng có thể tự đặt tên mới cho một kiểu dữ liệu đã có bằng từ khoá typealias: typealias TênMới = Kiểu Ví dụ: typealias Nguyen = Int sẽ định nghĩa kiểu số nguyên tiếng Việt có tên là Nguyen và ta có thể dùng nó để khai báo kiểu dữ liệu nguyên như với Int .
10	Các kiểu dữ liệu tự định nghĩa	Đa dạng, phụ thuộc vào nhu cầu của người dùng (sẽ được trình bày ở các phần sau).

Bảng 1.2.2 Một số kiểu dữ liệu thông dụng trong Swift

Ngoài ra, trong Swift còn cho phép người dùng tự định nghĩa một số loại kiểu dữ liệu khác nữa như Array, Dictionaries, Structures, Classes phục vụ cho nhu cầu đa dạng về dữ liệu trong các ứng dụng iOS. Các kiểu dữ liệu này sẽ được làm rõ hơn trong các chương sau.

Khai báo biến, hằng trong Swift: Trong Swift có thể khai báo biến theo một trong hai cách dưới đây:

```
var variableName:<data type> = <optional initial value>
Ví dụ: var name:String = "Nguyen Van A"
```

Hoặc khai báo ẩn danh kiểu dữ liệu như dưới đây:

```
var variableName = <initial value>
Ví dụ: var name = "Nguyen Van A"
```

Lưu ý: Giá trị khởi tạo cho biến có thể có hoặc không. Trường hợp trong lúc khai báo không chỉ rõ kiểu của dữ liệu (khai báo dạng 2) thì bắt buộc phải khai báo giá trị ban đầu cho biến, căn cứ vào đó trình biên dịch sẽ ngầm xác định kiểu gần với giá trị khởi tạo cho biến nhất (tính năng type inference trong Swift): Ví dụ nếu khai báo **var a = 50** thì biến a sẽ được trình biên dịch xác định là biến loại Int; khai báo **var b = 10 + 0.15** thì trình biên dịch sẽ xác định biến **b** là kiểu Double.

Trong Swift, nếu các biến không thay đổi giá trị của nó trong suốt quá trình hoạt động của ứng dụng, khi đó biến nên được khai báo dưới dạng hằng sử dụng từ khoá **let** thay cho từ khoá **var**.

1.2.3 Biến kiểu Optional

Optional là một kiểu biến đặc biệt trong ngôn ngữ Swift. Trong các ứng dụng iOS, mọi biến sau khi khai báo đều cần phải khởi tạo trước khi dùng, nếu không trình biên dịch

sẽ bắt lỗi. Tuy nhiên, trong thực tế có nhiều trường hợp chúng ta không thể biết trước giá trị khởi tạo cho biến lúc biên dịch mà chỉ khi chạy chương trình mới xác định được. Nghĩa là biến đó hoặc là có một giá trị nào đó hoặc là nó có thể không tồn tại (**nil** hay **rỗng**). Với những biến kiểu này chúng ta cần khai báo chúng là kiểu **Optional** và trình biên dịch sẽ không yêu cầu phải khởi tạo biến trước nữa. Để khai báo một biến là kiểu Optional, chúng ta dùng một trong những cú pháp sau đây (Thêm dấu ? hoặc dấu ! vào sau kiểu dữ liệu của biến):

```
var bienOptionalInt: Int? // Biến Optional kiểu Int
var bienOptionalString: String? = nil // Biến Optional kiểu String tường minh
var bienOptionalFloat: Float! // Biến Optional kiểu Float (Xử lý khác khi dùng)
```

Khi một biến được khai báo là Optional, thì khi sử dụng biến sẽ không theo cách thông thường mà cần phải được **unwrap** trước khi có thể sử dụng giá trị của biến. Để unwrap một biến Optional hãy đặt sau nó một ký tự “!”. Ví dụ sau sẽ hướng dẫn cách sử dụng biến Optional:

```
var greeting: String?
greeting = "Hello every ones"
if greeting != nil {
    print(greeting!) // Xuất chuỗi ra màn hình Console
}
else {
    print("biến greeting không có giá trị: nil")
}
```

Dòng số 1: Khai báo biến greeting là một biến kiểu Optional. Dòng số 2 gán giá trị cho biến này (Dòng này có thể có hoặc không. Nếu không sẽ cho kết quả khác). Khi làm việc với biến Optional ta luôn phải kiểm tra xem nó có khác rỗng hay không (greeting != nil) và xử lý các trường hợp đó theo các cách khác nhau. Dòng số 4, khi biến khác nil ta sẽ hiển thị giá trị của biến ra màn hình Console của Xcode bằng lệnh print(greeting!), dấu “!” sau tên biến dùng để **unwrap** giá trị của biến Optional greeting trước khi sử dụng được giá trị đó và hiện ra màn hình.

Lưu ý: Nếu biến Optional được khai báo như dạng 3 (sử dụng dấu “!” thay vì “?”) thì khi sử dụng biến không cần **unwrap** giá trị của nó nữa (nó được **unwrap** tự động). Tuy nhiên, trong thực tế, nếu không cần thiết hiếm khi người ta khai báo dạng 3, vì lập trình viên sẽ không bị cảnh báo **unwrap** giá trị trước khi sử dụng, dẫn đến lập trình viên

“quên” mất đó là biến Optional và dễ rơi vào tình trạng cố gắng truy xuất biến nil trong lúc chạy chương trình và không kiểm tra trước đó dẫn đến chương trình có thể bị chèt.

Do vậy, đối với biến Optional, khi sử dụng luôn cần **phải kiểm tra xem biến đó có nil hay không** (giống như ví dụ trên) hoặc dùng cách dưới đây (Optional Binding):

```
var greeting: String?  
greeting = "Hello every ones"  
if let myGreeting = greeting {  
    print(myGreeting) // Xuất chuỗi ra màn hình Console  
}  
else {  
    print("biến greeting không có giá trị: nil")  
}
```

Ở dòng thứ 3, thay vì kiểm tra xem biến **greeting** có nil hay không thì ta khai báo biến hằng trung gian **myGreeting** và sử dụng nó để lấy giá trị biến Optional (Chú ý ở đây không cần phải **unwrap** biến **greeting** nữa).

1.2.4 Biến Tuples

Trong các ứng dụng iOS, nhiều khi biến cần dùng lại là một nhóm giá trị các biến đơn lẻ, khi đó biến kiểu Tuples (biến bộ) sẽ được sử dụng.

Khai báo biến tuples trong iOS theo một trong những cách sau:

```
let error501 = (501, "Not implemented") // error501 là một biến tuples có kiểu ẩn  
let error501: (Int, String) = (501, "Not implemented") // error501 là một biến tuples  
let error501 = (code: 501, description: "Not implemented") // error501 là tuples có tên  
let error501: (code: Int, description: String) = (501, "Not implemented")
```

Ví dụ 1 Khai báo theo cách 1:

```
let person = ("Nguyen Van A", "34, Linh Dong, Thu Duc", 0976555555)  
print("Ho va ten \(person.0)")  
print("Dia chi \(person.1)")  
print("So dien thoai \(person.2)")
```

Khi đó biến **person** là một kiểu tuples có 3 trường giá trị là Họ và tên, Địa chỉ và Số điện thoại. Khi biến tuples không có tên, ta có thể truy xuất từng trường thông qua chỉ số (các trường được đánh chỉ số từ 0 đến n-1).

Hoặc ta có thể gán biến **person** cho một tuples có tên và truy xuất thông qua tên đó:

```
let person = ("Nguyen Van A", "34, Linh Dong, Thu Duc", 0976555555)  
let (name, address, phone) = person
```

```

print("Ho va ten \name")
print("Dia chi \adress")
print("So dien thoai \phone")

```

Hoặc có thể định nghĩa rõ tên từng trường trong biến tuples theo dạng 3, 4:

```

let person: (name: String, adress: String, phone: Int)
person = ("Nguyen Van A", "34, Linh Dong, Thu Duc", 0976555555)
print("Ho va ten \person.name")
print("Dia chi \person.adress")
print("So dien thoai \person.phone")

```

1.2.5 Chuỗi thoát trong Swift

Trong Swift, khi xử lý chuỗi nhiều khi cần đưa những ký tự đặc biệt vào trong chuỗi đó (ví dụ ký tự xuống dòng, ký tự '\', ký tự "", ký tự ",...). Để có thể thực hiện được điều đó cần sử dụng các chuỗi thoát tương ứng:

STT	Chuỗi thoát	Mô tả
1	\\	Đưa ký tự đặc biệt '\' vào trong chuỗi
2	\n	Đưa ký tự xuống dòng vào trong chuỗi
3	\r	Đưa ký tự Về đầu dòng vào trong chuỗi
4	\t	Đưa ký tự tab vào trong chuỗi
5	\'	Đưa ký tự "" vào trong chuỗi
6	\"	Đưa ký tự "" vào trong chuỗi

Bảng 1.2.5. Chuỗi thoát trong Swift

1.2.6 Các loại toán tử trong ngôn ngữ Swift

Trong Swift có nhiều loại toán tử khác nhau:

a) Các phép toán số học

STT	Phép toán	Mô tả
1	+	a + b Phép cộng a và b
2	-	a - b Phép trừ a cho b
3	*	a * b Phép nhân a với b
4	/	a / b Phép chia a cho b
5	%	a % b Phép lấy phần dư của phép chia a cho b
6	++	a++ Phép tăng giá trị a lên 1
7	--	a-- Phép giảm giá trị a đi 1

b) Các phép so sánh

STT	Phép toán	Mô tả
1	==	a == b Kết quả phép so sánh là true nếu giá trị a bằng giá trị b, ngược lại sẽ là false .

2	$!=$	a \neq b Kết quả phép so sánh là true nếu giá trị của a khác với giá trị của b và ngược lại sẽ là false .
3	$>$	a $>$ b Kết quả phép so sánh là true nếu giá trị của a lớn hơn giá trị của b và ngược lại sẽ là false .
4	$<$	a $<$ b Kết quả phép so sánh là true nếu giá trị của a nhỏ hơn giá trị của b và ngược lại sẽ là false .
5	\geq	a \geq b Kết quả phép so sánh là true nếu giá trị của a lớn hơn hoặc bằng giá trị của b và ngược lại sẽ là false .
6	\leq	a \leq b Kết quả so sánh là true nếu giá trị của a nhỏ hơn hoặc bằng giá trị của b và ngược lại sẽ là false .

c) Các phép toán logic (Thực hiện trên các biến logic)

STT	Phép toán	Mô tả
1	$\&\&$	a $\&\&$ b Phép và logic (AND)
2	\parallel	a \parallel b Phép hoặc logic (OR)
3	!	!a Phép phủ định logic (NOT)

d) Các phép xử lý bit

Giả sử giá trị của biến a là 97 (tương ứng chuỗi nhị phân 0110 0001) và biến b là 13 (tương ứng chuỗi nhị phân 0000 1101), khi đó các phép xử lý bit sẽ có kết quả sau:

STT	Phép toán	Mô tả
1	$\&$	a $\&$ b Phép và theo từng bit (kết quả là 0000 0001)
2	$ $	a $ $ b Phép hoặc theo từng bit (kết quả là 0110 1101)
3	\wedge	a \wedge b Phép XOR theo từng bit (kết quả là 0110 1100)
4	\sim	\sim a Phép đảo theo từng bit (kết quả là 1001 1110)
5	$<<$	a $<< 2$ Phép dịch 2 bit sang trái (kết quả 0110 000100)
6	$>>$	a $>> 2$ Phép dịch 2 bit sang phải (kết quả 00 0110 00)

e) Các phép gán

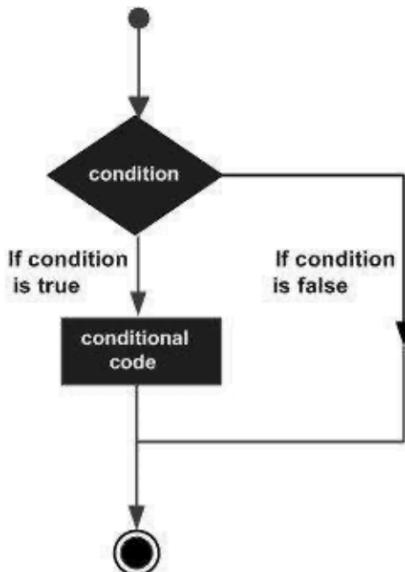
STT	Phép toán	Mô tả
1	$=$	c = a + b Kết quả phép a + b được gán cho biến c
2	$+=$	a $+=$ b Thực hiện giống như a = a + b
3	$-=$	a $-=$ b Thực hiện giống như a = a - b
4	$*=$	a $*=$ b Thực hiện giống như a = a * b
5	$/=$	a $/=$ b Thực hiện giống như a = a / b
6	$%=$	a $%=$ b Thực hiện giống như a = a % b

f) Các phép tập hợp

STT	Phép toán	Mô tả
1	a...b	1...5: Phạm vi là 1, 2, 3, 4, 5
2	a..<b	1..<5: Phạm vi là 1, 2, 3, 4
3	a...	1...: Phạm vi là 1, 2, 3, 4, 5, ...
4	...a	...100 Phạm vi là ..., 97, 98, 99, 100

1.2.7 Rẽ nhánh và vòng lặp trong Swift

Cú pháp rẽ nhánh loại 1: Câu lệnh if

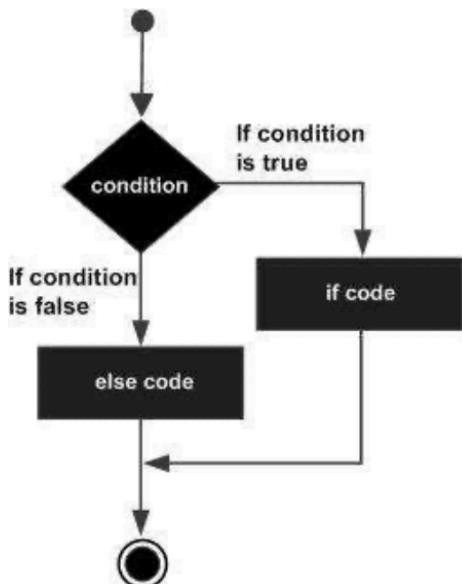


```
if <condition> {  
    conditional code  
}
```

Nếu biểu thức điều kiện **<condition>** đúng (biểu thức điều kiện nhận giá trị **true**) thì đoạn lệnh nằm trong **conditional code** sẽ được thực hiện. Ngược lại đoạn lệnh đó sẽ bị bỏ qua và không lệnh nào được thực hiện.

Hình 1.2.7.1 Hoạt động của câu lệnh rẽ nhánh loại 1

Cú pháp rẽ nhánh loại 2: Câu lệnh if...else



```
if <condition> {  
    <if code>  
}  
else {  
    <else code>  
}
```

Nếu biểu thức điều kiện **<condition>** nhận giá trị **true** (biểu thức điều kiện đúng) thì đoạn lệnh **<if code>** sẽ được thực hiện, ngược lại đoạn lệnh **<else code>** sẽ được thực hiện.

Hình 1.2.7.2 Hoạt động của câu lệnh rẽ nhánh loại 2

Cú pháp rẽ nhánh loại 3: Câu lệnh if ... else if ... else if ... else

Trường hợp có nhiều hơn 2 nhánh rẽ, chúng ta có thể sử dụng câu lệnh rẽ nhánh loại 3 theo cú pháp sử dụng như sau:

```
if <condition1> {  
    <condition1 code>  
}  
else {  
    ...  
    if <condition2> {  
        <condition2 code>  
    }  
    else {  
        ...  
        if <condition3> {  
            <condition3 code>  
        }  
        else {  
            ...  
            if <condition4> {  
                <condition4 code>  
            }  
            else {  
                ...  
            }  
        }  
    }  
}
```

```

else if <condition2> {
    <condition2 code>
}
else if <condition3> {
    <condition3 code>
}
...
else if <condition n> {
    <condition n code>
}
else {
    <other code>
}

```

Nếu biểu thức điều kiện <condition1> nhận giá trị true, thì đoạn lệnh tương ứng <condition1 code> sẽ được thực hiện, tương tự với **n-1** trường hợp còn lại. Riêng trường hợp cuối cùng đoạn lệnh trong <other code> sẽ được thực hiện khi và chỉ khi không có bất kỳ biểu thức điều kiện nào trước đó thoả mãn (từ <condition1> đến <condition n> đều nhận giá trị false).

Lưu ý: Các câu lệnh rẽ nhánh lại có thể được lồng trong nhau.

Cú pháp lệnh rẽ nhánh nhiều lựa chọn dạng switch

Trường hợp có nhiều hơn 2 nhánh rẽ, thường chúng ta sẽ dùng câu lệnh rẽ nhánh dạng 3 như ở trên hoặc sử dụng câu lệnh rẽ nhánh dạng switch theo cú pháp như sau:

```

switch <value> {
    case <value 1>:
        <Value 1 code>
    case <value 2>:
        <Value 2 code>
    ...
    case <value n>:
        <Value n code>
    default:
        <default code>
}

```

Khi dùng câu lệnh rẽ nhánh dạng switch thì biểu thức điều kiện <condition> được thay bằng một biểu thức giá trị <value>. Tùy theo giá trị cụ thể của biểu thức này, câu lệnh rẽ nhánh sẽ được thực hiện tương ứng: Nếu giá trị đó bằng <value 1> thì đoạn code trong <Value 1 code> sẽ được thực hiện. Tương tự với các trường hợp còn lại. Cuối

cùng, nếu không có trường hợp nào phù hợp thì đoạn lệnh trong <default code> sẽ được thực hiện (Rẽ nhánh này là tùy chọn, có thể có hoặc không). Câu lệnh **switch** trong ngôn ngữ Swift có đôi chút khác biệt với các ngôn ngữ khác như Java, C++... là trong các nhánh **không cần** sử dụng câu lệnh **break** để thoát khỏi nhánh hiện tại.

Một số ví dụ về các câu lệnh rẽ nhánh trong Swift

Ví dụ 1: Tạo một Playground và nhập vào đoạn code sau đây. Thay đổi giá trị của biến **diem** từ 0-10 mỗi lần chạy và xem kết quả. Thay đổi giá trị biến **diem** ngoài phạm vi giá trị từ 0-10 và xem kết quả.

```
import UIKit
var diem = 6
if diem >= 0 && diem < 4 {
    print("Diem \(diem) la loai Yeu")
}
else if diem >= 4 && diem < 7 {
    print("Diem \(diem) la loai Trung binh")
}
else if diem >= 7 && diem < 8 {
    print("Diem \(diem) la loai Kha")
}
else if diem >= 8 && diem < 9 {
    print("Diem \(diem) la loai Gioi")
}
else if diem >= 9 && diem <= 10 {
    print("Diem \(diem) la loai Xuat sac")
}
else {
    print("Diem \(diem) nam ngoai thang diem 0-10 cua Viet Nam")
}
```

Ví dụ 2: Thực hiện lại yêu cầu trong ví dụ 1 bằng câu lệnh rẽ nhánh switch. Thay đổi giá trị biến **diem** như ở ví dụ trên mỗi khi chạy chương trình và so sánh kết quả.

```
import UIKit
var diem = 6.1
switch diem {
case 0..<4:
    print("Diem \(diem) la loai Yeu")
case 4..<7:
    print("Diem \(diem) la loai Trung binh")
case 7..<8:
    print("Diem \(diem) la loai Kha")
case 8..<9:
    print("Diem \(diem) la loai Gioi")
case 9...10:
    print("Diem \(diem) la loai Xuat sac")
default:
    print("Diem \(diem) nam ngoai thang diem 0-10 cua Viet Nam")
}
```

Vòng lặp trong ngôn ngữ Swift

Trong ngôn ngữ Swift có 3 loại vòng lặp chính đó là vòng lặp xác định **for-in** (có số bước lặp xác định trước), vòng lặp không xác định **while** và **repeat ... while** (chưa biết trước số bước lặp mà nó phụ thuộc vào giá trị biểu thức điều kiện lúc chạy chương trình).

Vòng lặp xác định **for-in**

Cú pháp sử dụng:

```
for <item> in <items> {  
    <code>  
}
```

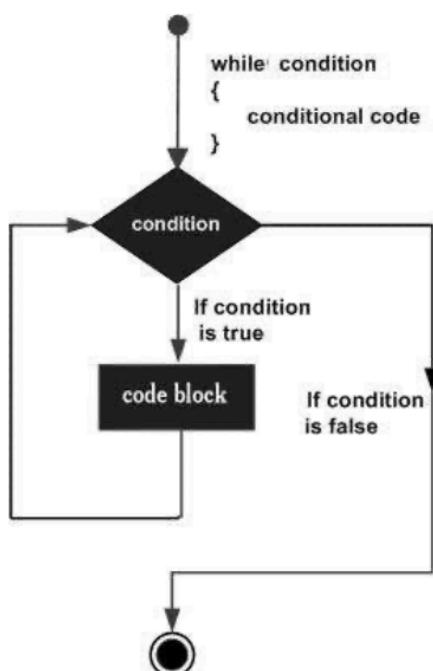
Đoạn chương trình trong `<code>` sẽ được thực hiện đến khi biến `<item>` không còn nằm trong tập hợp `<items>` nữa.

Ví dụ 3: thực hiện in ra màn hình các giá trị từ 0 đến 1000

```
import UIKit  
for i in 0...1000 {  
    print(i)  
}
```

Vòng lặp không xác định **while**

Cú pháp sử dụng:



Khi biểu thức điều kiện **condition** còn đúng, thì các câu lệnh trong **condition code** (hay chính là **code block** trong diagram Hình 1.2.7.3) còn được thực hiện. Vòng lặp chỉ dừng lại khi biểu thức điều kiện **condition** ở một bước lặp nào đó có giá trị **false**. Như vậy, vòng lặp while trước tiên sẽ kiểm tra biểu thức điều kiện nếu **đúng** các câu lệnh trong **code block** được thực hiện 1 lần, rồi chương trình lại quay lại kiểm tra biểu thức điều kiện nếu đúng nó lặp lại các công việc giống như trước, nếu sai chương trình thoát ra khỏi vòng lặp.

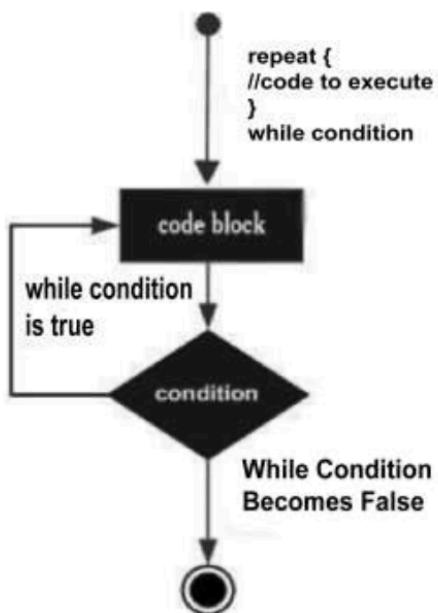
Hình 1.2.7.3 Hoạt động vòng lặp while

Ví dụ 4: Thực hiện in ra tất cả các giá trị từ 0 đến 10 sao cho giá trị trước nhỏ hơn giá trị sau 0.25. Lưu ý: Nếu biểu thức điều kiện không bao giờ sai chương trình sẽ rơi vào vòng lặp vô tận (không bao giờ thoát ra khỏi vòng lặp).

```
import UIKit
var index = 0.0
while index <= 10 {
    print(index)
    index += 0.25
}
```

Vòng lặp không xác định repeat ... while

Cú pháp sử dụng:



Vòng lặp **repeat ... while** hoạt động khác so với vòng lặp **while** ở trên. Vòng lặp này sẽ thực hiện **code block** trước sau đó mới kiểm tra điều kiện lặp. Như vậy vòng lặp loại này thì bao giờ **code block** cũng được thực hiện ít nhất 1 lần. Sau khi thực hiện xong **code block** lần thứ nhất, vòng lặp mới kiểm tra điều kiện lặp **condition** sau **while**, nếu điều kiện còn đúng chương trình tiếp tục quay lại thực hiện các lệnh trong **code block**, nếu sai chương trình thoát khỏi vòng lặp.

Hình 1.2.7.4 Hoạt động vòng lặp repeat ... while

Ví dụ 5: Thực hiện lại yêu cầu trong ví dụ 4 nhưng sử dụng vòng lặp repeat ... while.

```
import UIKit
var index = 0.0
repeat {
    print(index)
    index += 0.25
} while index <= 10
```

Lưu ý: Trong các vòng lặp có thể sử dụng các lệnh điều khiển như **break** và **continue** giống như ngôn ngữ Java hay C/C++, ... để thay đổi cách thực hiện lệnh bên trong vòng lặp (**break** sẽ thoát khỏi vòng lặp ngay còn **continue** sẽ bỏ qua các câu lệnh sau nó và bắt đầu vòng lặp tiếp theo).

1.2.8 Xử lý chuỗi và ký tự trong Swift

String trong Swift là một tập hợp có thứ tự các ký tự. Cũng như nhiều ngôn ngữ lập trình khác, thao tác xử lý chuỗi String cũng là một trong những công việc thường gặp.

Khai báo chuỗi

Có một số cách khai báo chuỗi String khác nhau như dưới đây:

```
var strA = "Hello World!" // Nếu trong dấu "" không có ký tự thì được chuỗi rỗng  
var strB: String = "Hello World!"  
var strC = String("Hello World!") // Nếu strC = String() thì nó là chuỗi rỗng  
var char1: Character = "A" // Khai biến kiểu ký tự  
var char2 = "B" // Khai báo biến ký tự ẩn kiểu  
var char3: Character = "" // Sai vì không thể khai báo ký tự rỗng hoặc hơn 1 ký tự.  
Lưu ý: Nếu muốn khai báo một chuỗi trên nhiều dòng thì đặt các dòng đó vào giữa """.
```

Một số thao tác xử lý trên chuỗi thông dụng

STT	Thao tác xử lý	Mô tả, ví dụ
1	Kiểm tra chuỗi rỗng	var strA = String("Hello World!") if !strA.isEmpty { print("Chuoi khong rong va co gia tri \(strA)") }
2	Nối chuỗi	var strB = "Hello TDC" let strC = strA + strB // strC chứa chuỗi ghép var name = "Cuong" let greeting = "Xin chao ban \(name)"//Ghép chuỗi với biến
3	Đếm số ký tự	let num = strB.count // Trả về số ký tự có trong chuỗi strB
4	So sánh chuỗi == (bằng) và != (khác)	if strA == strB { // Kiểm tra khác nhau dùng != print("Hai chuoi bang nhau.") }
5	Đưa ký tự Unicode vào chuỗi	let strD = "Ky tu Unicde: \u{00BC}"
6	Duyệt chuỗi	var strA = "Hello World!" for char in strA { print(char, terminator: " ") // Mỗi ký tự cách nhau " ". }
7	Kiểm tra tiền tố	var strA = "Hello World!" strA.hasPrefix("He") // Cho kết quả true vì strA có tiền tố là chuỗi "He"
8	Kiểm tra hậu tố	var strA = "Hello World!" strA.hasSuffix("ld!") // Cho kết quả true vì strA có chứa hậu tố là chuỗi "ld!".

9	Chuyển chuỗi thành số nguyên, số thực	var strB = "12345" let num: Int = Int(strB) // Cách 1 hoặc let num: Int = (strB as NSString).integerValue // Cách 2 Với số thực làm tương tự.
10	So sánh <	strA < strB So sánh theo thứ tự chữ cái từng ký tự. Ví dụ nếu var strA = "Hello World!" var strB = "Hello Cuong!" thì khi đó strA < strB cho kết quả là false vì đến ký tự khác biệt là W so với C thì W lớn hơn.
11	Truy xuất chuỗi	Trong Swift, để truy xuất từng phần tử của chuỗi cần truy xuất thông qua chỉ số của chuỗi. Các hàm dưới đây cho phép lấy các chỉ số chuỗi để truy xuất: var strA = "Hello World!" let chiSoKyTuDau = strA.startIndex let kyTuDauTien = strA[chiSoKyTuDau] // Là ký tự 'H' let chiSoKyTuCuoi = strA.index(before: strA.endIndex) let kyTuCuoi = strA[chiSoKyTuCuoi] // Là ký tự '!' let chiSoKyTuThu7 = strA.index(strA.startIndex, offsetBy: 6) let kyTuThu7 = strA[chiSoKyTuThu7] // Là ký tự 'W'
12	Chèn thêm và cắt bớt trong chuỗi	var strA = "Hello World!" var strB = "Hello Cuong!" // Ghép chuỗi strB vào ngay sau strA strA.insert(contentsOf: strB, at: strA.endIndex) // Chèn thêm 1 ký tự '!' vào cuối strA strA.insert("!", at: strA.endIndex) // Xoá bỏ ký tự cuối cùng trong strA strA.remove(at: strA.index(before: strA.endIndex)) // Xoá bỏ từ ký tự trắng đến cuối chuỗi let subRange = strA.index(strA.startIndex, offsetBy: 5)...strA.index(before: strA.endIndex) strA.removeSubrange(subRange)
13	Nối biến ký tự vào trong chuỗi	Không thể dùng toán tử + để nối trực tiếp biến ký tự vào trong chuỗi được. Muốn nối ta dùng cách sau: var strA = "Hello World" var char = "!" strA = strA + char // Không được phép strA.append(char) // Ok strA = "\u{0028}strA\u{0029}\u{0028}char\u{0029}" // Ok

Bảng 1.2.8.1 Một số thao tác xử lý chuỗi thông dụng

1.2.9 Mảng trong Swift

Khai báo mảng

var arrayName = [Kieu]() // Cách 1: Khai báo một mảng rỗng kiểu **Kieu**.

var nums = [Int]() // Ví dụ khai báo một mảng rỗng tên **nums** kiểu Int.

// Cách 2: Khai báo mảng tên là **arrayName** gồm N phần tử kiểu **Kieu** và khởi gán giá // trị **GiaTri** cho mỗi phần tử của mảng đó.

```

var arrayName = [Kieu](repeating: GiaTri, count: N)
// Khai báo mảng names gồm 10 phần tử và gán giá trị cho mỗi phần tử là "New name"
var names = [String](repeating: "New name", count: 10)
// Cách 3: Khai báo và khởi gán mảng arrayName với các phần tử là giatri1, giatri2, ...
// có kiểu Kieu hoặc kiểu ẩn (các giá trị cùng kiểu). Lưu ý, nếu các giá trị giatri1, giatri2,
// giatri3, ... khác kiểu nhau thì Kieu phải được chỉ rõ là Any.
var arrayName = [giatri1, giatri2, giatri3,...] hoặc
var arrayName:[Kieu] = [giatri1, giatri2, giatri3,...]
// Ví dụ khai báo mảng nguyên tên arrayInt có 3 phần tử 10, 30 và 9
var arrayInt: [Int] = [10, 30, 9]

```

Truy xuất mảng trong Swift

Cũng giống cấu trúc dữ liệu mảng trong các ngôn ngữ lập trình khác như C, C++, C#, Java,... trong Swift mỗi phần tử của mảng cũng có một chỉ số (*index*) dùng để truy xuất giá trị của nó (đánh số từ 0 đến n-1 cho mảng gồm **n** phần tử) và cách truy xuất cũng giống những ngôn ngữ lập trình đó. Với mảng arrayInt ở ví dụ trên ta có thể:

```

arrayInt[0] = 100 // Thay đổi giá trị phần tử số 1 của mảng từ 10 thành 100
print("Gia tri phan tu so 2 cua mang la: \(arrayInt[1])") // Lấy giá trị phần tử số 2

```

Thêm phần tử vào mảng, nối mảng, duyệt mảng

Trong Swift để có thể thêm phần tử mới vào mảng có thể dùng hàm `append()`, hoặc dùng toán tử nối mảng ‘+=’ và toán tử ‘+’:

```

import UIKit
var nums = [Int]()
nums.append(10) // Thêm phần tử 10 vào mảng
nums.append(20) // Thêm phần tử 20 vào mảng
nums += [30] // Thêm phần tử 30 vào mảng
nums += [40, 50, 60] // Nối mảng [10, 20, 30] với mảng [40, 50, 60]
for item in nums { // Duyệt mảng dùng for-in
    print(item)
}

```

Sử dụng chỉ số của các phần tử khi duyệt mảng với for-in

Trong Swift, đôi khi trong lúc duyệt mảng chúng ta cần biết chỉ số của mỗi phần tử trong mảng đó. Khác với những ngôn ngữ lập trình khác, với for-in thông thường, ta chưa thể

biết chính xác chỉ số của phần tử đang được duyệt. Để làm được điều đó ta thực hiện như sau (sử dụng phương thức `enumerated()` của mảng cho ví dụ ở trên):

```
import UIKit
var nums = [Int]()
nums.append(10)
nums.append(20)
nums += [30]
nums += [40, 50, 60]
// Phương thức enumerated() trả lại cho mỗi phần tử của mảng về một bộ hai số là
// vị trí của phần tử đó trong mảng index và giá trị của phần tử đó value.
for (index, value) in nums.enumerated() {
    print("Phan tu thu \(index+1) cua mang la: \(value)")
}
```

Lưu ý: Cũng giống mảng chuỗi ký tự (String), có thể dùng thuộc tính `.count` để đếm số phần tử trong mảng; dùng thuộc tính `.isEmpty` để kiểm tra xem mảng có rỗng hay không.

1.2.10 Tập hợp trong Swift

Cũng giống một số ngôn ngữ lập trình khác, tập hợp (Set) trong Swift được dùng để lưu trữ các giá trị **khác nhau** của cùng một kiểu dữ liệu, nhưng khác với mảng, các phần tử của tập hợp thì không có thứ tự (Mảng thì mỗi phần tử luôn có thứ tự index nhất định). Do vậy, sử dụng tập hợp thay cho mảng trong những trường hợp mà thứ tự các phần tử trong đó không quan trọng và các giá trị của chúng **khác biệt nhau** (không muôn có những giá trị bị trùng lặp trong đó).

Khai báo tập hợp (Set)

```
var setName = Set<Kieu>() // Khai báo một tập hợp rỗng kiểu Kieu hoặc
var setName: Set = [value1, value2, ...] // Khai báo và khởi gán giá trị cho tập hợp hoặc
var setName: Set<Kieu> = [value1, value2, ...] // Khai báo và khởi gán với kiểu Kieu.
```

Một số phép xử lý thường thực hiện trên cấu trúc kiểu tập hợp

Ví dụ với khai báo sau ta có thể thực hiện:

```
var setInt = Set<Int>()
```

STT	Phép toán	Mô tả
1	<code>setInt.isEmpty</code>	Kiểm tra tập hợp có rỗng không (Trường hợp này kết quả là true)
2	<code>setInt.insert(10)</code>	Đưa một phần tử vào trong tập hợp (Với tập hợp số nguyên <code>setInt</code> ở trên thì phần tử 10 được thêm vào). Kết quả phép thêm vào sẽ trả về bộ (inserted,

		memberAfterInsert), trong đó inserted . Nếu phép thêm vào thành công thì inserted sẽ là true, ngược lại sẽ nhận giá trị false, còn memberAfterInsert sẽ chứa các phần tử của tập hợp sau phép thêm.
3	<code>setInt.count</code>	Trả về số các phần tử trong tập hợp
4	<code>setInt.remove(10)</code>	Xoá phần tử có giá trị 10 trong tập hợp setInt. Nếu tập hợp có chứa giá trị đó, phần tử sẽ bị xoá và phép xoá trả về chính phần tử đó. Ngược lại không phần tử nào bị xoá và nó trả về nil.
5	<code>setInt.contains(10)</code>	Kiểm tra sự tồn tại của phần tử trong tập hợp, nếu có trả về true, ngược lại trả về false.
6	<code>for item in setInt {...}</code>	Duyệt tập hợp sử dụng for-in. Lưu ý, trong cách duyệt này thì các phần tử được duyệt sẽ không theo một thứ tự nhất định nào.
7	<code>for item in setInt.sorted() {...}</code>	Duyệt tập hợp theo thứ tự được sắp xếp.
8	<code>.union .intersection .subtracting</code>	Phép union sẽ trả về một tập hợp mới là kết quả của phép hợp giữa hai tập hợp. Phép intersection sẽ trả về một tập hợp mới là kết quả của phép giao giữa hai tập hợp. Còn phép subtracting sẽ trả về một tập hợp mới là kết quả của phép loại trừ các phần tử của tập hợp số 2 ra khỏi tập hợp số 1.

Một số ví dụ về tập hợp

Ví dụ 1: Thực hiện các phép union, intersection và subtracting

```
import UIKit
let setOdds: Set = [1,3,5,7,9] // Tập hợp số lẻ => { 3, 5, 7, 1, 9 }
let setEvens: Set = [2,4,6,8] // Tập hợp số chẵn => { 6, 4, 2, 8 }
let primes: Set = [2,3,5,7] // Tập hợp số nguyên tố => { 2, 7, 3, 5 }
setOdds.union(setEvens) // => { 7, 8, 9, 6, 2, 3, 1, 4, 5 }
setOdds.intersection(setEvens) // => {} Rỗng
setOdds.intersection(primes) // => { 3, 7, 5 }
setOdds.subtracting(primes) // => { 9, 1 }
setOdds.union(setEvens).subtracting(primes) // => { 4, 6, 9, 8, 1 }
```

Ví dụ 2: Phát sinh n số ngẫu nhiên **khác nhau** cho một ứng dụng iOS.

```
import UIKit
// Tạo tập hợp rỗng kiểu Int
var setRandom = Set<Int>()
// Thực hiện vòng lặp đến khi đủ 100 số ngẫu nhiên
while setRandom.count < 100 {
    // Phát sinh số ngẫu nhiên trong khoảng [1,1000]
    let numRandom = Int.random(in: 1...1000)
    // Đảm bảo số ngẫu nhiên sẽ không bị lặp lại
    if setRandom.insert(numRandom).inserted {
        // Sử dụng số ngẫu nhiên vừa tạo
        print("So ngau nhien hop le tiep theo \(numRandom)")
    }
}
```

1.2.11 Làm việc với kiểu Dictionaries

Dictionary là một kiểu tổ chức dữ liệu đặc trưng trong Swift dùng để lưu trữ danh sách các phần tử không tính thứ tự (khác với mảng) của cùng một kiểu dữ liệu. Mỗi phần tử của Dictionary được định danh bởi một **key** (khoá) để truy xuất sau này (Khác với Set).

Khai báo Dictionary

```
var dicName = [KeyType:ValueType]() // Khai báo một Dictionary rỗng, hoặc
```

```
var dicName:[KeyType:ValueType] = [key1:value1, key2:value2, ...] // Khởi gán
```

```
var dicName:Dictionary = [key1:value1, key2:value2, ...] // Khai báo và khởi gán
```

Truy xuất biến Dictionary

Truy xuất biến dictionary thông qua từ khoá key tương ứng: ví dụ **dicName[key2]** của khai báo trên đây sẽ trả về giá trị tương ứng trong dictionary là **value2**. Lưu ý: Giá trị trả về khi truy xuất biến dictionary có thể có hoặc nil nên nó là giá trị Optional. Ví dụ: Nếu ta dùng cấu trúc Dictionary để biểu diễn số dân của các quận trong Thành phố Hồ Chí Minh thì đoạn chương trình sẽ như sau (trong đó key của dictionary là tên mỗi quận, và giá trị tương ứng chính là dân số của quận đó):

```
import UIKit
var danSo:[String:Int] = ["Thu Duc": 1000000, "Quan 1": 2000000, "Quan 2": 1500000]
print("So dan trong quan Thu Duc la \(danSo["Thu Duc"]!)")
```

Kết quả khi chạy sẽ cho ra: **So dan trong quan Thu Duc la 1000000**

Để thay đổi giá trị của một phần tử trong Dictionary có thể gán giá trị trực tiếp hoặc sử dụng hàm **.updateValue()** thông qua giá trị của khoá (key) tương ứng. Với ví dụ trên để thay đổi dân số của quận Thủ Đức ta có thể:

```
danSo["Thu Duc"] = 10000000 // Gán giá trị trực tiếp cho phần tử của Dictionary
// Hoặc dùng hàm. Hàm sẽ trả về nil hoặc giá trị bị thay thế. Biến danSoCu sẽ là 1000000
let danSoCu = danSo.updateValue(10000000, forKey: "Thu Duc")
```

Để xoá một phần tử từ Dictionary có thể dùng **.removeValue(forKey:)** hoặc gán trực tiếp giá trị **nil** cho phần tử đó. Ví dụ xoá thông tin về dân số quận Thủ Đức:

```
danSo.removeValue(forKey: "Thu Duc") // Trả về số dân bị xoá hoặc nil
// Hoặc gán trực tiếp giá trị nil
danSo["Thu Duc"] = nil
```

Để duyệt các phần tử trong Dictionary dùng for-in với bộ giá trị (key, value):

```
for (key, value) in danSo {  
    print("Dan so cua quan \\" + key + "\ la: " + value)  
}
```

Và kết quả chạy chương trình sẽ là (Thứ tự có thể khác sau các lần chạy):

```
Dan so cua quan Thu Duc la: 1000000  
Dan so cua quan Quan 1 la: 2000000  
Dan so cua quan Quan 2 la: 1500000
```

Mảng và Dictionary

Đây là thao tác thường được sử dụng nhiều trong thực tiễn nhằm biến những mảng đã có thành kiểu Dictionary để dễ quản lý và ngược lại. Hãy xem một số ví dụ dưới đây:

Ví dụ 1: Với ví dụ trên cấu trúc Dictionary sẽ rất hữu ích khi cần biết số dân của từng quận trong Thành phố Hồ Chí Minh. Tuy nhiên, nếu chỉ muốn lấy danh sách tên các quận hoặc số liệu dân số các quận (không cần biết tên cụ thể quận nào) thì thao tác chuyển đổi Dictionary thành mảng lại rất hữu ích. Câu lệnh sau sẽ tách tên các quận trong Dictionary thành mảng kiểu **String** và số liệu dân số các quận thành mảng **Int**:

```
let danhSachQuan = [String](danSo.keys) // ["Quan 2", "Thu Duc", "Quan 1"]  
let phanBoDanSoCacQuan = [Int](danSo.values) // [1500000, 1000000, 2000000]
```

Ví dụ 2: Quản lý khoảng cách địa lý giữa các thành phố.

```
import UIKit  
// Mảng chứa danh sách tên các thành phố  
var cities = ["Hà Nội", "Hồ Chí Minh", "Hải Phòng", "Đà Nẵng", "Nam Định"]  
// Mảng chứa khoảng cách giữa các thành phố tương ứng ở trên với trung tâm  
var distance = [0, 2000, 130, 1000, 120]  
// Dictionary chứa thông tin thành phố và khoảng cách với trung tâm  
let cityDistances = Dictionary(uniqueKeysWithValues: zip(cities, distance))
```

Hoặc dùng lệnh sau cũng cho kết quả tương tự câu lệnh cuối ở trên:

```
let cityDistances = [String:Int](uniqueKeysWithValues: zip(cities, distance))
```

Trong đó, hàm **zip(mảng1, mảng2)** sẽ tạo ra mảng gồm các cặp **value1:value2** với value1 lấy từ mảng1 và value2 lấy từ mảng2 tương ứng. Cuối cùng tham số **uniqueKeysWithValues** nhằm đảm bảo rằng mỗi cặp **value1:value2** là duy nhất với **value1** sẽ là **key** của Dictionary **cityDistances**. Và kết quả của Dictionary này là:

```
[{"Đà Nẵng": 1000, "Hà Nội": 0, "Hồ Chí Minh": 2000, "Hải Phòng": 130, "Nam Định": 120]
```

1.2.12 Lệnh guard

Lệnh **guard** là câu lệnh có thể dùng thay thế hữu hiệu cho các lệnh điều kiện if ... else trong một số trường hợp nhất định. Cú pháp của lệnh **guard** như sau:

```
guard <condition> else {  
    <statements>  
}  
<other statements>
```

Nếu điều kiện **<condition>** thoả mãn thì các lệnh sau khối lệnh **guard** **<other statements>** sẽ được tiếp tục thực hiện. Ngược lại, các lệnh **<statements>** trong khối lệnh **else** của **guard** sẽ được thực hiện (các lưu trữ, thông báo, ... và kết thúc bằng lệnh thoát khỏi ngữ cảnh hiện tại: hàm, chương trình... bằng một trong những lệnh sau: return, break, continue, throw). Do vậy, nếu các điều kiện trong **<condition>** **không thoả mãn** thì những lệnh phía sau khối **guard** là **<other statements>** sẽ không bao giờ được thực hiện. Ví dụ dưới đây sẽ làm rõ hơn cách dùng **guard** so với câu lệnh điều kiện điều khiển rẽ nhánh **if...else**:

```
guard let username = usernameField?.text,  
       let password = passwordField?.text,  
       !username.isEmpty, !password.isEmpty else {  
    throw ...  
}
```

Với lệnh **guard** trên đây, có 4 điều kiện được kiểm tra. Nếu **username** là **nil** HOẶC **password** là **nil** HOẶC **username** rỗng HOẶC **password** rỗng thì khối lệnh **else** sẽ được thực hiện và các lệnh sau khối **guard** sẽ bị bỏ qua. Ngược lại, nếu **unwrap** được biến **usernameField.text** VÀ biến **passwordField.text** VÀ cả hai biến được **unwrap** là **username** và **password** đều khác rỗng thì các lệnh sau khối **guard** được thực hiện. Trường hợp này nếu dùng **if...else** biểu thức điều kiện sẽ phức tạp hơn nhiều, trong đó vừa phải **unwrap** các biến **Optional** (**if let** vòng ngoài), vừa phải kiểm tra điều kiện rỗng của hai biến **password** và **username** (**if...else** vòng trong) thì mới thực hiện được.

1.2.13 Hàm trong Swift

Cũng như nhiều ngôn ngữ lập trình khác, hàm là tập hợp những câu lệnh có tổ chức nhằm thực hiện một (một vài) nhiệm vụ xác định trong chương trình. Nhờ có hàm mà

cấu trúc chương trình sẽ rõ ràng hơn và ta có thể xây dựng chương trình từ tập hợp nhiều mô đun chương trình nhỏ hơn, dễ thực hiện hơn và có khả năng tái sử dụng cho các mục đích khác nhau. Về cơ bản, khi làm việc với hàm cần quan tâm đến 3 vấn đề: **Khai báo hàm, định nghĩa hàm; Cách truyền tham số vào/ra hàm; và Cách sử dụng hàm.**

Định nghĩa hàm trong Swift

Trong Swift, để định nghĩa hàm ta dùng từ khoá **func** với cú pháp sau:

```
func <tenHam>(<CacThamSo>) -> <KieuGiaTriTraVe> {  
    <CacLenhTrongThanHam>  
}
```

Trong đó **tenHam** do người lập trình đặt theo quy tắc đặt tên biến, **CacThamSo** là các giá trị truyền **vào/ra** của hàm, **KieuGiaTriTraVe** là kiểu của giá trị lấy ra khỏi hàm và **CacLenhTrongThanHam** là toàn bộ các câu lệnh thực hiện chức năng của hàm để từ các tham số đầu vào (input) tính toán, xử lý... theo thuật toán nào đó và cho kết quả đầu ra (output) theo yêu cầu của hàm (dùng lệnh return trong thân hàm để trả kết quả output).

Ví dụ 1: Viết hàm trả về kết quả là phép nhân của hai giá trị nguyên a và b bất kỳ.

```
import UIKit  
// Định nghĩa hàm  
func mul(a:Int, b:Int) -> Int {  
    return a*b  
}  
// Lời gọi hàm  
let result = mul(a: 10, b: 20)
```

Trong ví dụ trên, hàm **mul** đã được định nghĩa với: Input là hai giá trị kiểu Int là a và b và Output là một giá trị kiểu Int. Hàm sẽ thực hiện phép nhân a với b ($a * b$) và trả lại kết quả cho nơi gọi nó bằng câu lệnh return ($return a * b$). Như vậy sau lời gọi hàm thì **result** sẽ chứa kết quả trả về của hàm **mul** (chính là $a * b$).

Lưu ý: Nếu hàm không có giá trị trả về (hoặc không có tham số) thì có thể bỏ [$->$ **<KieuGiaTriTraVe> ()**] (hoặc danh sách tham số **<CacThamSo>**) trong định nghĩa của hàm đó. Ví dụ hàm sau không có giá trị đầu vào (input), không có giá trị đầu ra (output) và chỉ làm duy nhất việc hiển thị lời chào “Hello” ra màn hình:

```
func chao() {  
    print("Hello")  
}
```

Truyền tham số vào/ra và Sử dụng hàm trong Swift

Trước tiên ta làm quen với khái niệm **Tên ngoài** và **Tên trong** (khác biệt với các ngôn ngữ lập trình khác) trong định nghĩa hàm của ngôn ngữ swift. Hãy để ý Định nghĩa hàm và Lời gọi hàm trong ví dụ 1: Trong định nghĩa hàm có hai tham số là a và b thì trong lời gọi hàm sẽ ghi tường minh `mul(a: 10, b: 20)` để người sử dụng biết chắc giá trị truyền vào 10 là cho tham số a và giá trị truyền vào 20 là cho tham số b (tránh nhầm lẫn thứ tự tham số khi gọi hàm trong các ngôn ngữ khác). Tên tham số được dùng trong thân hàm (chỗ câu lệnh `return a*b`) được gọi là **Tên trong**, còn tên tham số được dùng ở lời gọi của hàm được gọi là **Tên ngoài**. Trong ngôn ngữ lập trình Swift thì mặc định **Tên trong** sẽ trùng với **Tên ngoài** (như ở ví dụ trên). Trường hợp ta muốn có **Tên trong**, nhưng không muốn có **Tên ngoài** (tức lời gọi hàm không cần đưa chỉ dẫn a: và b: vào nữa) thì khi đó ta định nghĩa hàm như sau:

```
import UIKit
// Định nghĩa hàm
func mul(_ a:Int, _ b:Int) -> Int {
    return a*b
}
// Lời gọi hàm
let resultInt = mul(10, 20)
```

Trường hợp ta không muốn dùng **Tên ngoài** và **Tên trong** như mặc định nữa mà tự định nghĩa **Tên ngoài** khác thì ta làm như sau:

```
import UIKit
// Định nghĩa hàm
func mul(giaTriA a:Int, giaTriB b:Int) -> Int {
    return a*b
}
// Lời gọi hàm
let resultInt = mul(giaTriA: 10, giaTriB: 20)
```

Trường hợp này **giaTriA** và **giaTriB** là các **Tên ngoài** tương ứng với các tham số a và b (**Tên trong**) của định nghĩa hàm.

Việc truyền tham số cho các hàm trong Swift (cũng giống các ngôn ngữ khác) có thể chia thành hai loại là truyền theo **tham biến** và truyền theo **tham trị**. Để dễ hiểu hơn chúng ta có thể phân biệt: Truyền theo tham trị là trường hợp ta muốn đưa giá trị từ bên ngoài vào trong hàm thông qua các tham số của hàm và không cần lấy giá trị ra từ chúng. Còn truyền theo **tham biến** thì ngược lại, ta vừa muốn đưa các giá trị từ bên ngoài vào

trong hàm (input của hàm) vừa muốn lấy giá trị đã được tính toán trong hàm ra bên ngoài **quá các tham số của hàm** (output). Với hai cách truyền này thì mỗi ngôn ngữ lập trình sẽ có cú pháp và cách giải quyết khác nhau. Với các ví dụ ta vừa thực hiện cho hàm **mul**, thì các tham số a, b là những tham số đầu vào và chúng không thể lấy giá trị đầu ra của hàm, do vậy cách truyền chính là truyền theo **tham trị**. Tuy nhiên, mặc dù ta không dùng tham số a, b để lấy giá trị đầu ra của hàm nhưng ta vẫn có thể lấy được kết quả của phép $a*b$ từ bên trong thân hàm ra bên ngoài thông qua **giá trị trả về của hàm** (through qua câu lệnh return trong hàm mul). Nhiều trường hợp, hàm có nhiều giá trị trả về và phức tạp thì việc lấy các giá trị đó ra khỏi hàm thường thông qua cách truyền theo **tham biến**.

Ví dụ 2: Viết hàm thực hiện việc hoán vị giá trị cho hai biến a và b. Như vậy a, b sẽ là hai giá trị đầu vào cho hàm (để hàm thực hiện việc hoán đổi chúng ở trong thân hàm) và kết quả đầu ra của hàm (sau khi hàm thực hiện việc hoán đổi) vẫn chính là hai tham số a, b này (trường hợp này ta không thể dùng đưa giá trị a, b ra bên ngoài hàm bằng câu lệnh return được). Do đó, trường hợp này ta sẽ sử dụng truyền theo **tham biến** a, b cho hàm **hoanVi**. Để thực hiện điều đó, ta làm như sau (Ta dùng tên ngoài với mục đích cho lời gọi hàm sau này rõ ràng hơn):

```
func hoanVi(giaTriA a: inout Int, giaTriB b: inout Int) {
    let temp = a
    a = b
    b = temp
}
```

Trước khai báo kiểu dữ liệu của mỗi tham số ta đặt thêm từ khoá **inout** nhằm báo cho trình biên dịch biết rằng tham số a, b là các **tham biến** (vừa đưa giá trị vào, vừa lấy giá trị ra khỏi hàm). Khi đó, các tham số a và b không phải là tham số cục bộ của hàm nữa mà nó sẽ tham chiếu đến hai biến a và b khác bên ngoài hàm và mọi thay đổi trong hàm đều giữ lại khi thoát khỏi hàm. Lời gọi hàm trong trường hợp này sẽ là:

```
var a = 10
var b = 20
hoanVi(giaTriA: &a, giaTriB: &b)
```

a, b là hai **biến bên ngoài hàm** (phải định nghĩa bằng **var**) và được truyền vào trong hàm với ký tự tham chiếu ‘&’ (Nếu truyền theo tham trị với trường hợp này, thì giá trị

từ bên ngoài được copy vào hai **biến hằng a và b cục bộ trong hàm** và trong thân hàm sẽ phát sinh lỗi vì biến hằng không thể thay đổi – không thể hoán vị giá trị của chúng).

Truyền tham số kiểu generic trong Swift

Trong **ví dụ 1** ở trên, rõ ràng hàm **mul** chỉ có thể thực hiện phép nhân cho các số a và b có kiểu dữ liệu Int. Nếu muốn hàm trả kết quả phép nhân cho các giá trị có kiểu bất kỳ (Float, Double, ...) ta cần định nghĩa hàm kiểu **generic** với **T là kiểu Numeric bất kỳ** và tham số truyền vào cũng như giá trị trả về sẽ định nghĩa theo T như sau:

```
import UIKit
// Định nghĩa hàm
func mul<T: Numeric>(a:T, b:T) -> T {
    return a*b
}
// Lời gọi hàm
let resultInt = mul(a: 10, b: 20)
let resultDouble = mul(a: 20.2, b: 10)
```

Tương tự, ta có thể viết lại hàm **hoanVi** trong ví dụ 2 để có thể hoán vị cho bất kỳ biến với bất kỳ kiểu dữ liệu nào:

```
func hoanVi<T>(giaTriA a: inout T, giaTriB b: inout T) {
    let temp = a
    a = b
    b = temp
}
var a = "Hello"
var b = "Chao"
hoanVi(giaTriA: &a, giaTriB: &b)
print("Loi chao a: \(a), b: \(b)") // Loi chao a: Chao, b: Hello
```

Trường hợp đặc biệt, nếu số lượng tham số đưa vào hàm không biết trước, ta có thể khai báo như sau:

Ví dụ 3: Viết hàm thực hiện tính tổng của n số bất kỳ.

```
import UIKit
func genericSum<T: Numeric>(_ numbers: T...) -> T {
    var sum:T = 0
    for num in numbers {
        sum += num
    }
    return sum
}
let result1 = genericSum(1, 3, 5, 7, 9.1) // 25.1
let result2 = genericSum(20.2, 3, 60, 70.4, 6, 5.5, 9) // 174.1
```

Lưu ý: Khai báo “...” sau kiểu của tham số (giống với Java) là chìa khoá ví dụ này.

Biến Kiểu hàm

Một biến không nhất thiết chỉ được phép chứa giá trị cho các kiểu dữ liệu nhất định, mà chúng còn có thể “trỏ” đến một hàm khác. Khi đó biến phải được khai báo kiểu là một **Kiểu hàm** (prototype của hàm) tương thích với hàm nó sẽ “trỏ” tới. Kiểu hàm được viết theo cú pháp: **(Kiểu tham số1, Kiểu tham số 2, ... Kiểu tham số n)-> Kiểu trả về.**

Với hàm **mul** ở ví dụ 1 thì **Kiểu hàm** của nó sẽ là: (Int, Int)->Int; với hàm hoanVi trong ví dụ 2 thì Kiểu hàm của nó sẽ là: (inout Int, inout Int)->Void...

```
func hoanVi(giaTriA a: inout Int, giaTriB b: inout Int) {
    let temp = a
    a = b
    b = temp
}
// doiCho là một biến hàm và nó trả về hàm hoanVi
var doiCho: (inout Int, inout Int)-> Void = hoanVi
var a = 10
var b = 20
doiCho(&a, &b) // Cho cùng kết quả giống như gọi hoanVi(&a, &b)
```

Trong nhiều trường hợp, tham số cho hàm cũng có thể là một hàm khác. Khi đó ta cần dùng biến hàm làm tham số của hàm.

Ví dụ 4: Viết hàm có thể thực hiện các phép cộng, trừ, nhân, chia của hai số nguyên bất kỳ a, b với các hàm add, sub, mul và div đã được định nghĩa từ trước. Như vậy ta cần định nghĩa một hàm có khả năng nhận tên của một hàm khác làm tham số đầu vào (sử dụng biến kiểu hàm **mathFunc**) và hai giá trị a, b cần tính:

```
import UIKit
func add(a: Int, b: Int) -> Int {
    return a + b
}
func sub(a: Int, b: Int) -> Int {
    return a - b
}
func mul(a: Int, b: Int) -> Int {
    return a * b
}
func div(a: Int, b: Int) -> Int {
    return a / b
}
func calculate(mathFunc: (Int, Int)->Int, a: Int, b: Int) -> Int {
    return mathFunc(a, b)
}
// Lời gọi hàm
calculate(mathFunc: add, a: 10, b: 20)
```

```
calculate(mathFunc: sub, a: 10, b: 20)
calculate(mathFunc: mul, a: 10, b: 20)
calculate(mathFunc: div, a: 10, b: 20)
```

1.2.14 Định nghĩa và sử dụng Cấu trúc, Lớp đối tượng trong Swift

Cũng giống như nhiều ngôn ngữ lập trình hướng đối tượng khác việc định nghĩa **lớp** và sử dụng các **đối tượng** trong Swift đều tuân thủ các nguyên tắc và đặc trưng cơ bản của phương pháp tiếp cận hướng đối tượng. Trong phần này sẽ không nhắc lại các kiến thức về lập trình hướng đối tượng mà đi vào cụ thể cú pháp định nghĩa và cách sử dụng các đối tượng trong Swift. Tương tự với định nghĩa và sử dụng **Cấu trúc**.

Lớp và Cấu trúc có nhiều điểm tương đồng và cũng có nhiều khác biệt mà chỉ lớp mới có, còn cấu trúc thì không. Các điểm tương đồng giữa lớp và cấu trúc là:

- Đều định nghĩa các thuộc tính và phương thức của chúng
- Đều định nghĩa các hàm khởi tạo
- Đều có thể mở rộng chức năng bằng từ khoá extension

Một số điểm khác biệt chỉ có ở lớp đối tượng đó là:

- Tính kế thừa của lớp đối tượng
- Khả năng ép kiểu
- Hàm huỷ để giải phóng tài nguyên
- Một đối tượng có thể có nhiều tham chiếu tại một thời điểm

Định nghĩa lớp trong Swift

Dùng cú pháp sau:

```
class <ClassName>: <Super class> {
    <code for class's body>
}
```

Trong đó: <ClassName> là tên của lớp đang định nghĩa; <Super class> là các lớp mà lớp đang định nghĩa sẽ kế thừa (hoặc các nhóm phương thức mà lớp sẽ được uỷ quyền thực hiện giống như interface trong Java), nếu nhiều lớp được kế thừa sẽ phân tách bởi dấu ‘,’; Và <code for class's body> sẽ là đoạn chương trình nhằm định nghĩa các thuộc tính và phương thức bên trong lớp đó.

Định nghĩa cấu trúc trong Swift

Dùng cú pháp sau:

```
struct <structName> {
    <fields>
}
```

Trong đó: <structName> là tên của cấu trúc và <fields> sẽ là đoạn chương trình định nghĩa các thuộc tính và phương thức của cấu trúc.

Khởi tạo (Initialization)

Trong ngôn ngữ Swift mọi thuộc tính trong class hay struct sau khi khai báo nếu không phải kiểu Optional thì bắt buộc phải khởi tạo giá trị (Hoạt động giống Constructor trong Java hoặc C++). Cú pháp của hàm khởi tạo như sau:

```
init(<parameters>) {  
    <statements>  
}
```

Trong đó: <parameters> là các tham số truyền vào hàm khởi tạo và <statements> là những câu lệnh căn cứ vào các giá trị truyền vào để khởi gán giá trị cho từng thuộc tính của lớp hoặc cấu trúc.

Ví dụ 1: Định nghĩa và sử dụng cấu trúc Resolution và lớp VideoMode như dưới đây.

```
import UIKit  
// Cấu trúc Resolution lưu độ phân giải của Video  
struct Resolution {  
    var width: Int  
    var height: Int  
    init(width: Int, height: Int) {  
        self.width = width  
        self.height = height  
    }  
}  
class VideoMode { // Lớp đối tượng VideoMode  
    var resolution: Resolution  
    var interlaced: Bool  
    var frameRate: Double  
    var name: String?  
    // Hàm khởi tạo các thuộc tính cho lớp đối tượng  
    init(resolution: Resolution, interlaced: Bool, frameRate: Double) {  
        self.resolution = resolution  
        self.interlaced = interlaced  
        self.frameRate = frameRate  
    }  
}  
let videoResolution = Resolution(width: 640, height: 480)  
let videoMode = VideoMode(resolution: videoResolution, interlaced:  
false, frameRate: 25.0) //videoMode là một đối tượng (instance) của lớp VideoMode  
Lưu ý: biến hằng videoResolution ở trên cũng giống như các biến (biến hằng) từ các  
kiểu dữ liệu cơ bản hoặc mảng, hoặc dictionary trong Swift chúng đều là biến (biến  
hằng) thuộc loại Value Types (Kiểu giá trị), nghĩa là mỗi khi chúng được gán cho biến  
khác hoặc truyền vào trong hàm thì giá trị của chúng được copy lại. Ngược lại, biến
```

hằng videoMode là đối tượng của một lớp nên chúng thuộc loại **Reference Types** (Kiểu tham chiếu), nghĩa là tại một thời điểm có thể có nhiều tham chiếu đến nó (Phép gán hay truyền tham số cho hàm sẽ tham chiếu đến chính đối tượng đó) và do đó mặc dù biến tham chiếu là biến hằng thì ta vẫn có thể thay đổi giá trị của đối tượng mà nó tham chiếu đến (vì nó tham chiếu chứ không chứa đối tượng đó). Trong ví dụ trên, câu lệnh sau hoàn toàn hợp lệ: `videoMode.interlaced = true.` Tuy nhiên, câu lệnh sau là không hợp lệ: `videoResolution.width = 1080` (vì biến `videoResolution` là biến hằng và **thuộc loại value types**).

Phép so sánh định danh đối tượng trong Swift

Do một đối tượng tại một thời điểm có thể có nhiều tham chiếu đến nó, nên nếu muốn xác định xem hai tham chiếu có cùng trỏ đến một đối tượng hay không ta **không** thể dùng phép so sánh thông thường là `==` hoặc `!=` (Hai toán tử này dùng cho các biến thuộc loại value types, riêng với struct cần tự định nghĩa) mà phải dùng: `===` (So sánh bằng) và `!==` (So sánh khác). Với ví dụ trên nếu ta thêm lệnh `let v = videoMode` thì khi đó phép so sánh (`v === videoMode`) sẽ cho kết quả `true` và (`v !== videoMode`) sẽ cho kết quả `false` (Vì thực chất `v` và `videoMode` đều là hai biến tham chiếu đến cùng đối tượng).

Phép so sánh bằng

Nếu muốn so sánh bằng về mặt giá trị hai biến thể (instances) **khác nhau** của một cấu trúc hoặc một lớp thì chúng ta cần tự định nghĩa lại phép so sánh đó (`==` hoặc `!=`). Với các định nghĩa ở **ví dụ 1** trên đây, ta có thể mở rộng chức năng cho chúng như sau:

```
extension Resolution: Equatable {
    static func == (left: Resolution, right: Resolution) -> Bool {
        return (left.height == right.height) && (left.width ==
            right.width)
    }
    static func != (left: Resolution, right: Resolution) -> Bool {
        return !(left == right)
    }
}
extension VideoMode: Equatable {
    static func == (left: VideoMode, right: VideoMode) -> Bool {
        return (left.resolution == right.resolution) &&
            (left.interlaced == right.interlaced) && (left.frameRate ==
                right.frameRate) && (left.name == right.name)
    }
    static func != (left: VideoMode, right: VideoMode) -> Bool {
        return !(left == right)
    }
}
```

```

}
let anotherVideoResolution = videoResolution
let anotherVideoMode = VideoMode(resolution: videoResolution,
interlaced: true, frameRate: 30)
// So sánh bằng cho hai biến thế khác nhau của struct => true
var equal = (videoResolution == anotherVideoResolution)
// So sánh bằng cho hai đối tượng khác nhau của lớp => false
equal = (videoMode == anotherVideoMode)

```

1.2.15 Ép kiểu

Khác so với nhiều ngôn ngữ lập trình hướng đối tượng, Swift có thể ép kiểu theo hai chiều. Để hiểu rõ cơ chế ép kiểu trong Swift, xét các lớp trong ví dụ sau:

```

import UIKit
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}
class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
// MediaItem Array
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]

```

Kiểm tra kiểu với “is”: Mảng library trên đây được Swift biên dịch thành mảng các MediaItem cho dù các đối tượng được tham chiếu bên trong nó vẫn là Movie và Song. Do đó, khi sử dụng chúng cần nhận diện xem chúng thuộc lớp đối tượng nào (Movie hay Song) sử dụng “is”.

```

// Kiểm tra kiểu các phần tử trong mảng dùng is
var movieCount = 0
var songCount = 0

for item in library {

```

```

    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

```

Ép kiểu dạng 1: Downcasting dùng “as?”

```

// Ép kiểu Downcasting
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

```

Với dạng ép kiểu này do biến tham chiếu **item** có thể trở đến nhiều loại đối tượng thuộc lớp con (Subclass) của nó (Movie hoặc Song), nên kết quả trả về của phép ép kiểu có thể có (ép đúng loại đối tượng) hoặc nil (ép sai loại đối tượng). Nghĩa là kết quả phép ép kiểu loại này là một biến Optional (nên ở đây ta unwrap nó bằng if ... let).

Lưu ý: Thứ nhất, nếu chúng ta đảm bảo chắc chắn rằng phép ép kiểu luôn trả về kết quả đúng (nếu không khi chạy sẽ báo lỗi), khi đó ta có thể sử dụng “**as!**” (vừa thực hiện ép kiểu vừa unwrap kết quả nhận được) thay cho “**as?**”. Thứ hai, bản thân đối tượng sau khi ép kiểu không hề thay đổi.

Ép kiểu dạng 2: Ép kiểu dùng “as”

Bổ sung thêm đoạn lệnh sau vào ví dụ trên:

```

// Mảng hỗn hợp nhiều loại phần tử khác nhau
var things = [Any]()
things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })

```

Trong Swift cung cấp hai loại kiểu dữ liệu đặc biệt là **Any** và **AnyObject**. Any có thể biểu diễn cho bất cứ biến thể của bất cứ kiểu dữ liệu nào, kể cả kiểu hàm. Còn AnyObject có thể biểu diễn cho bất kỳ đối tượng nào. Mảng **things** ở trên là một mảng hỗn hợp

chứa 2 phần tử số nguyên, 2 phần tử số thực, 1 phần tử chuỗi, 1 phần tử kiểu tuple (Double, Double), 1 đối tượng kiểu Movie và một dạng biến hàm (dưới dạng Closure). Tuy nhiên, nếu đưa một biến kiểu Optional vào mảng Swift vẫn cảnh báo. Để tránh điều đó ta nên ép kiểu **tường minh** biến Optional về dạng Any trước khi đưa vào mảng:

```
let optionalNumber: Int? = 3
things.append(optionalNumber)           // Warning
things.append(optionalNumber as Any) // No warning
```

Khi sử dụng mảng hỗn hợp ở trên, ta cần ép kiểu chúng cho phù hợp:

```
// Khi sử dụng mảng hỗn hợp cần ép kiểu về dạng đúng của từng phần tử
for thing in things {
    switch thing {
        case 0 as Int:
            print("zero as an Int")
        case 0 as Double:
            print("zero as a Double")
        case let someInt as Int:
            print("an integer value of \(someInt)")
        case let someDouble as Double where someDouble > 0:
            print("a positive double value of \(someDouble)")
        case is Double:
            print("some other double value that I don't want to print")
        case let someString as String:
            print("a string value of \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("an (x, y) point at \(x), \(y)")
        case let movie as Movie:
            print("a movie called \(movie.name), dir. \(movie.director)")
        case let stringConverter as (String) -> String:
            print(stringConverter("Michael"))
        default:
            print("something else")
    }
}
```

1.2.16 Protocol và cơ chế Delegate trong Swift

Protocol trong Swift là một **kiểu** dùng để định nghĩa cho các chức năng (không định nghĩa việc lưu trữ dữ liệu), hay nói cách khác, Protocol là một cách để chúng ta định nghĩa các API cho phép nơi gọi tự định nghĩa chức năng mà họ mong muốn. Điều này giúp chương trình linh hoạt hơn giống như việc dùng các interfaces trong ngôn ngữ Java.

Cú pháp định nghĩa cho Protocol có dạng:

```
protocol <protocolName> {
    <requirements>
}
```

Trong đó <requirements> là định nghĩa prototype của các hàm (API) cho protocol. Ví dụ dưới đây sẽ định nghĩa một Protocol chứa hàm **random** của một Generator. Hàm này không nhận đối truyền vào và trả về một giá trị Double (chỉ là dạng prototype):

```
protocol RandomNumberGenerator {  
    func random() -> Double  
}
```

Tùy nhu cầu sử dụng mà ở các lớp kế thừa nó sẽ tự định nghĩa ra các loại hàm sinh số ngẫu nhiên khác nhau theo các thuật toán khác nhau. Ví dụ, ta có thể xây dựng một lớp có tên **LinearCongruentialGenerator** để tạo số ngẫu nhiên có giá trị trong khoảng từ 0 đến 1 (nhỏ hơn 1) theo thuật toán đồng dư tuyến tính (Linear congruential generator):

```
// Algorithm linear congruential generator  
class LinearCongruentialGenerator: RandomNumberGenerator {  
    var lastRandom = 42.0  
    let m = 139968.0  
    let a = 3877.0  
    let c = 29573.0  
    // Thực hiện chức năng hàm random  
    func random() -> Double {  
        lastRandom = ((lastRandom * a + c)  
            .truncatingRemainder(dividingBy:m))  
        return lastRandom / m  
    }  
    let generator = LinearCongruentialGenerator()  
    print("Here's a random number: \(generator.random())")  
    // Prints "Here's a random number: 0.3746499199817101"  
    print("And another one: \(generator.random())")  
    // Prints "And another one: 0.729023776863283"
```

Lưu ý: Nếu muốn các API của protocol có thể thay đổi các biến thể của lớp định nghĩa nó thì cần đưa từ khoá **mutating** vào trước **func**.

Cơ chế Delegate (Uỷ quyền) trong Swift

Trong Swift, nhiều thư viện đối tượng được xây dựng sẵn để sử dụng. Tuy nhiên, nhiều chức năng của chúng lại không thể xây dựng trước (do nhu cầu người dùng là khác nhau). Cơ chế uỷ quyền sẽ giúp ta giải quyết vấn đề này.

Xét ví dụ sau: Viết chương trình mô phỏng mối quan hệ giữa cửa hàng sản xuất bánh quy và cửa hàng bán bánh quy ra thị trường. Trước tiên chúng ta định nghĩa cấu trúc dữ liệu của một chiếc bánh quy như sau:

```
struct Cookie {  
    var size:Int = 5  
    var hasChocolateChips:Bool = false  
}
```

Tiếp theo, chúng ta định nghĩa cửa hàng sản xuất bánh quy như sau:

```
class Bakery
{
    func makeCookie()
    {
        var cookie = Cookie()
        cookie.size = 6
        cookie.hasChocolateChips = true
    }
}
```

Lớp này có một hàm để tạo ra những chiếc bánh quy theo yêu cầu. Tuy nhiên, để bán những chiếc bánh này lại không phải chức năng của cửa hàng làm bánh mà là chức năng của cửa hàng bán bánh. Điều này gây khó khăn cho việc tổ chức chương trình vì mỗi khi bánh được làm xong, nó cần được bán nhưng ta lại không thể định nghĩa chức năng này trong cửa hàng sản xuất bánh được, vì đó là chức năng của cửa hàng bán bánh. Cơ chế uy quyền sẽ giải quyết vấn đề này.

Trước tiên, ta cần định nghĩa **Protocol** có chứa chức năng bán của cửa hàng bán bánh:

```
protocol BakeryDelegate {
    func sellingCookie(_ cookie: Cookie)
}
```

Và hàm sellingCookie() sẽ được gọi mỗi khi cửa hàng làm bánh sản xuất xong. Để thực hiện điều đó, chúng ta điều chỉnh lại định nghĩa cho cửa hàng làm bánh như sau:

```
class Bakery
{
    var delegate:BakeryDelegate?
    func makeCookie()
    {
        var cookie = Cookie()
        cookie.size = 6
        cookie.hasChocolateChips = true

        delegate?.sellingCookie(cookie)
    }
}
```

Trong đó có hai điều chỉnh: Thêm một biến delegate có kiểu là Protocol mới định nghĩa ở trên (có chứa chức năng bán bánh) và gọi chức năng bán bánh này mỗi khi cửa hàng sản xuất xong. Nhưng ở đây cửa hàng bán bánh không cần biết bánh sẽ được bán như thế nào (nghĩa là lớp này không có trách nhiệm định nghĩa hàm bán bánh hoạt động ra sao) mà nó sẽ ủy quyền việc đó cho cửa hàng bán bánh thực hiện. Định nghĩa như sau:

```
class CookieShop: BakeryDelegate // Lớp cửa hàng bán bánh
{
    func sellingCookie(_ cookie: Cookie)
    {
        print("Buy cookie, with size \((cookie.size)!)")
    }
}
```

Cuối cùng chạy chương trình như sau:

```
let shop = CookieShop() // Tạo cửa hàng bán bánh
let bakery = Bakery()   // Tạo cửa hàng làm bánh
bakery.delegate = shop // Uỷ quyền bán cho shop
bakery.makeCookie()    // Thực hiện sản xuất bánh
```

Mỗi khi bánh được sản xuất xong thì nó sẽ được bán ngay bởi cửa hàng shop và dòng sau đây sẽ hiện trên màn hình kết quả: **Buy cookie, with size 6!**

1.3 Câu hỏi và bài tập chương 1

1. Khi đoạn code sau được thực hiện, giá trị biến `hàng j` là bao nhiêu?

```
let i = "5"
let j = i + i
```

2. Cho biết đoạn code sau sẽ cho kết quả thế nào?

```
let names = ["Chris", "Joe", "Doug", "Jordan"]
if let name = names[1] {
    print("Brought to you by \(name)")
}
```

3. Đoạn code sau đây sẽ cho kết quả thế nào?

```
func sayHello(to name: String) -> String {
    return "Howdy, \(name)!"
}
print("\(sayHello(to: "Jayne"))")
```

4. Khi thực hiện đoạn code sau sẽ thu được gì?

```
for i in 3...1 {
    print(i)
}
```

5. Cho biết kết quả thực hiện các câu lệnh sau đây là gì?

```
struct Starship {
    var name: String
}
let tardis = Starship(name: "TARDIS")
var enterprise = tardis
enterprise.name = "Enterprise"
print(tardis.name)
```

6. Khi thực hiện đoạn chương trình sau, kết quả thu được là gì?

```
let names = ["Serenity", "Sulaco", "Enterprise", "Galactica"]
for name in names where name.hasPrefix("S") {
    print(name)
```

```
}
```

7. Cho biết sau khi thực hiện đoạn code sau ta thu được gì?

```
var motto = "Bow ties are cool"  
motto.replacingOccurrences(of: "Bow", with: "Neck")  
print(motto)
```

8. Hãy cho biết nếu thực hiện đoạn code sau đây, ta thu được gì?

```
final class Dog {  
    func bark() {  
        print("Woof!")  
    }  
}  
  
class Corgi : Dog {  
    override func bark() {  
        print("Yip!")  
    }  
}  
  
let muttface = Corgi()  
muttface.bark()
```

9. Hãy cho biết biến hàng third sẽ có giá trị thế nào nếu thực hiện đoạn lệnh sau?

```
let first = ["Sulaco", "Nostromo"]  
let second = ["X-Wing", "TIE Fighter"]  
let third = first + second
```

10. Hãy cho biết kết quả khi thực hiện đoạn chương trình sau?

```
let i = 3  
switch i {  
case 1:  
    print("Number was 1")  
case 2:  
    print("Number was 2")  
case 3:  
    print("Number was 3")  
}
```

11. Viết hàm thực hiện phép nhân cho n số bất kỳ!

12. Cho biết kết quả thực hiện đoạn code sau đây?

```
class Starship {  
    var name: String  
}  
  
let tardis = Starship(name: "TARDIS")
```

```
var enterprise = tardis
enterprise.name = "Enterprise"
print(tardis.name)
```

13. Cho biết kết quả thực hiện đoạn code sau? Hãy so sánh với câu 12 và câu 5!

```
class Starship {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let tardis = Starship(name: "TARDIS")
var enterprise = tardis
enterprise.name = "Enterprise"
print(tardis.name)
```

14. Sau khi thực hiện đoạn chương trình sau, ta thu được gì? Giải thích!

```
func sum(numbers: Int...) -> Int {
    var result = 0
    for number in numbers {
        result += number
    }
    return result
}
let result = sum(numbers: [1, 2, 3, 4, 5])
```

15. Cho biết kết quả nếu thực hiện đoạn chương trình sau?

```
var crew = ["Captain": "Malcolm", "Doctor": "Simon"]
crew = [:]
print(crew.count)
```

16. Cho biết kết quả nếu thực hiện đoạn chương trình sau? Giải thích!

```
let point = (556, 0)
switch point {
    case (let x, 0):
        print("X was \(\x)")
    case (0, let y):
        print("Y was \(\y)")
    case let (x, y):
        print("X was \(\x) and Y was \(\y)")
}
```

17. Cho biết kết quả nếu thực hiện đoạn chương trình sau? Giải thích!

```
func greet(names: String...) {
```

```

        print("Criminal masterminds:", names.joined(separator: ", "))
    }
greet(names: "Malcolm", "Kaylee", "Zoe")

```

18. Cho biết nếu thực hiện đoạn chương trình sau sẽ thu được kết quả gì? Giải thích!

```

struct Spaceship {
    var name: String
    func setName(_ newName: String) {
        name = newName
    }
}
var enterprise = Spaceship(name: "Enterprise")
enterprise.setName("Enterprise A")
print(enterprise.name)

```

19. Cho biết nếu chạy đoạn chương trình sau, chuyện gì sẽ xảy ra? Giải thích!

```

class Starship {
    var name: String
    override init(initialName: String) {
        name = initialName
    }
}
let serenity = Starship(initialName: "Serenity")
print(serenity.name)

```

20. Cho biết nếu chạy đoạn chương trình sau, chuyện gì sẽ xảy ra? Giải thích!

```

let name = "Simon"
switch name {
    case "Simon":
        fallthrough
    case "Malcom", "Zoe", "Kaylee":
        print("Crew")
    default:
        print("Not crew")
}

```

21. Cho biết biến kết quả (result) khi chạy đoạn chương trình sau? Giải thích!

```

let names: [String?] = ["Barbara", nil, "Janet", nil, "Peter", nil,
    "George"]
let result = names.compactMap { $0 }

```

22. Nếu thực hiện đoạn chương trình sau đây thì kiểu của biến testVar là gì? Giải thích!

```

let names = ["Pilot": "Wash", "Doctor": "Simon"]
for (key, value) in names.enumerated() {

```

```
    let testVar = value  
}
```

23. Cho biết kết quả nếu chạy đoạn chương trình sau? Giải thích!

```
let status = "shiny"  
for (position, character) in status.reversed().enumerated() where  
position % 2 == 0 {  
    print("\(position): \(character)")  
}
```

24. Cho biết kết quả nếu chạy đoạn chương trình sau? Giải thích!

```
struct User {  
    let name: String  
}  
  
let users = [User(name: "Eric"), User(name: "Maeve"), User(name:  
"Otis")]  
  
let mapped = users.map(\.name)  
print(mapped)
```

25. Hãy giải thích kết quả khi thực hiện đoạn chương trình sau?

```
struct TaylorFan {  
    static var favoriteSong = "Shake it Off"  
    var name: String  
    var age: Int  
}  
  
let fan = TaylorFan(name: "James", age: 25)  
print(fan.favoriteSong)
```

27. Nếu thực hiện đoạn chương trình sau sẽ có kết quả thế nào? Giải thích!

```
func square<T>(_ value: T) -> T {  
    return value * value  
}  
  
print(square(5))
```

28*. Hãy giải thích kết quả nếu thực hiện đoạn chương trình sau?

```
let i = 101  
  
if case 100...101 = i {  
    print("Hello, world!")  
} else {  
    print("Goodbye, world!")  
}
```

CHƯƠNG 2. THIẾT KẾ GIAO DIỆN VÀ XỬ LÝ SỰ KIỆN TRÊN IOS

Mục tiêu:

- Về kiến thức:
 - + Vận dụng được mô hình MVC vào phát triển ứng dụng trên iOS
 - Về kỹ năng:
 - + Thiết kế được các giao diện cơ bản cho các ứng dụng trên iOS;
 - + Hiện thực hóa các ứng dụng vừa và nhỏ trên iOS;
 - Về năng lực tự chủ và trách nhiệm:
 - + Luôn chủ động tìm hiểu vấn đề khi thực hiện các nhiệm vụ được giao
 - + Luôn tuân thủ đầy đủ các quy định của lớp học.

Mô tả nội dung: Sinh viên được học những kiến thức và kỹ năng trong xây dựng giao diện cho các ứng dụng trên iOS và xử lý các tương tác với người dùng. Các kiến thức, kỹ năng được truyền đạt trong giáo trình được sử dụng thuận nhất trên vài ứng dụng thực tiễn như một case study cụ thể.

2.1 Thiết kế giao diện với Storyboard

2.1.1 Các thành phần chính trong iOS Project

Sau khi tạo mới một Project trong Xcode, màn hình giao diện giống như sau xuất hiện:



Hình 2.1.1.1 Các thành phần chính của iOS Project

Màn hình giao diện chính có thể chia thành 4 vùng khác biệt: Navigation Area, Editor Area, Debug Area và Utility Area (Hình 2.1.1.1).

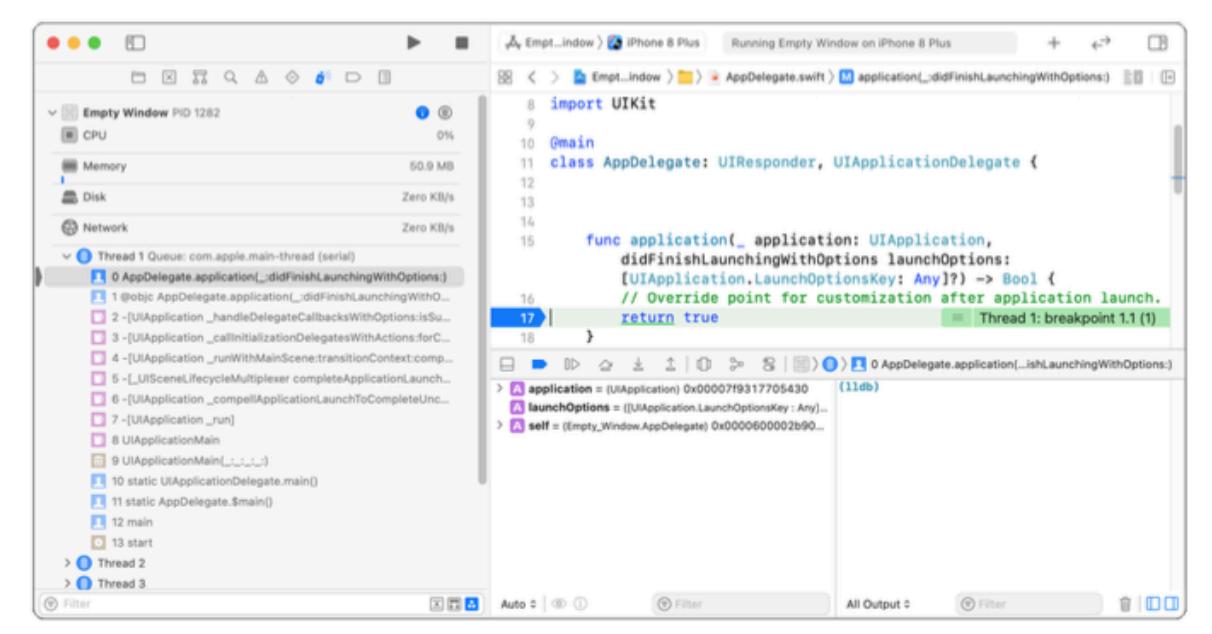
Navigation Area cho phép lập trình viên điều hướng đến các file hoặc các chức năng khác nhau trong Xcode. Có 9 điều hướng quan trọng:

- **Project Navigator:** Cho phép thực hiện các thao tác trên các files của Project như mở file, thêm xoá file, gom nhóm files... để tổ chức code và viết chương trình.
- **Source Control Navigator:** Xem lịch sử thay đổi của các phiên bản của Project nếu đang sử dụng các phần mềm quản lý phiên bản code như SVN hay Git.
- **Symbol Navigator:** Xem cấu trúc cây tổ chức lớp của ứng dụng.
- **Find Navigator:** Công cụ cho phép tìm kiếm nhanh bên trong Project.
- **Issue Navigator:** Hiện tất cả các cảnh báo (Warnings), các lỗi (errors) mà trình biên dịch tìm thấy trong Project.
- **Test Navigator:** Công cụ cho phép tạo, quản lý, thực hiện và quan sát các Unit Test.
- **Debug Navigator:** Kiểm tra các tiến trình và các thông tin liên quan trong quá trình thực hiện chương trình.
- **Break Point Navigator:** Quản lý tất cả các điểm Breakpoint mà lập trình viên đã thiết lập trong khi debug chương trình trong toàn bộ Project.
- **Report Navigator:** Xem lịch sử biên dịch của Project.

Editor Area cho phép lập trình viên biên soạn các tài nguyên trong Project:

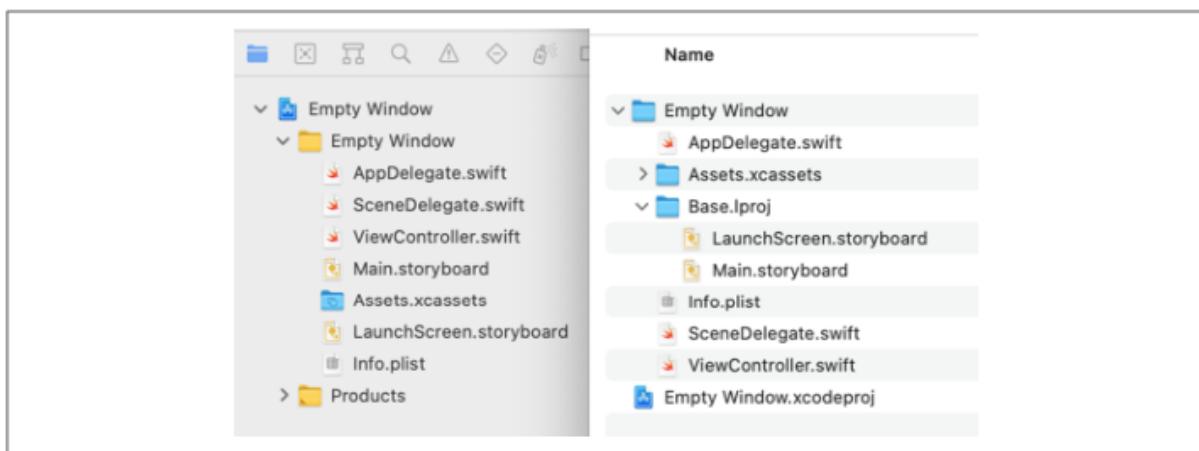
- **Source Editor:** Công cụ cho phép biên soạn và chỉnh sửa code trong ngôn ngữ Swift.
- **Interface Builder:** Công cụ cho phép lập trình viên xây dựng, điều chỉnh giao diện người dùng của ứng dụng iOS và gắn kết chúng với các Source code của Project. Để vào chức năng này, trên vùng Navigation chọn Main.storyboard (File chứa thiết kế giao diện của ứng dụng iOS).
- **Project Editor:** Cho phép xem và thiết lập cấu hình của Project iOS.

Debug Area là nơi xem vết chạy chương trình (call stack) với tên các phương thức đang thực hiện tại điểm dừng (Hình 2.1.1.2) và Utility Area là nơi chứa các thuộc tính của mỗi đối tượng được chọn.



Hình 2.1.1.2 Màn hình Debug chương trình trong iOS

Cấu trúc một iOS Project (Hình 2.1.1.3):



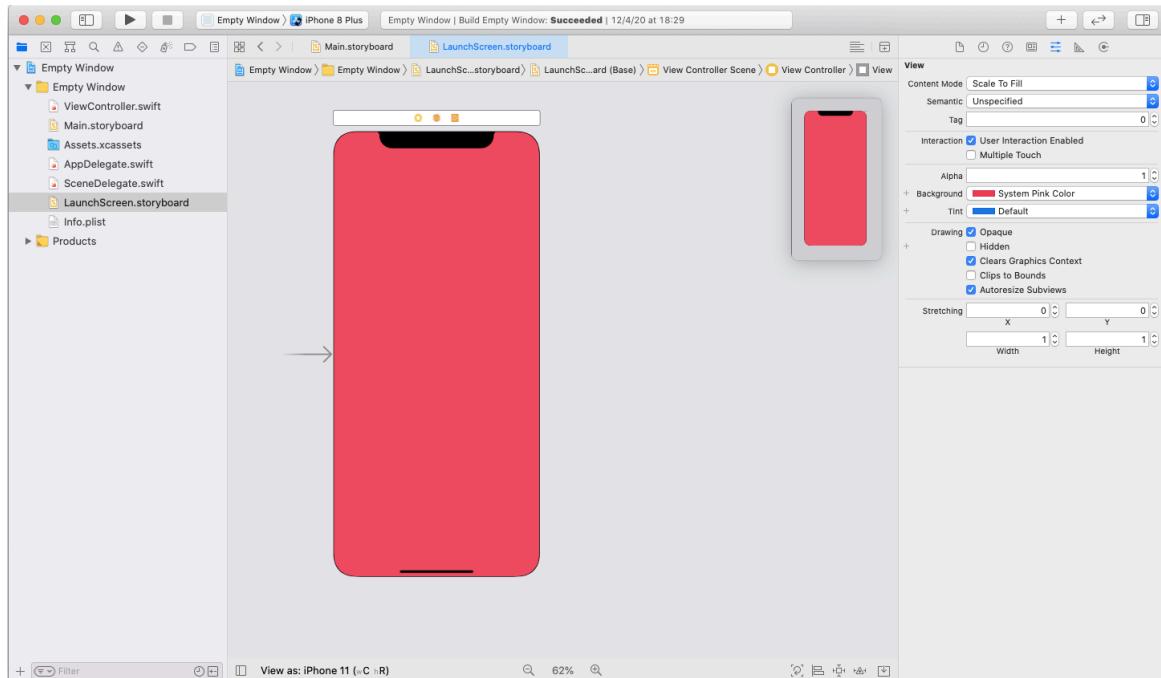
Hình 2.1.1.3 Cấu trúc iOS Project trong Xcode (trái) và trong thư mục dự án (phải)

- Các file dạng *.xcodeproj: Là file quan trọng chứa thông tin cấu trúc và cấu hình của Project. Nếu file này bị hỏng thì Project sẽ bị phá huỷ.
- Một số file hệ thống mặc định (tại thời điểm này chưa cần thay đổi) như AppDelegate.swift, ScenceDelegate.swift, Info.plist (File cấu hình hệ thống).

- File chứa cấu trúc giao diện người dùng *.storyboard. Trong đó Main.storyboard là giao diện chương trình còn LaunchScreen.storyboard là màn hình lúc khởi động.
- Assets.xcassets chứa các tài nguyên của ứng dụng như images, fonts, icons...
- ViewController.swift: File chứa code chương trình cho các màn hình cụ thể (trong ví dụ này ứng dụng chỉ có một màn hình duy nhất).
- Products: Nơi chứa các file biên dịch của ứng dụng iOS.

2.1.2 Màn hình chờ LaunchScreen.storyboard

Mỗi khi chạy một ứng dụng iOS sẽ có một khoảng thời gian chờ trước khi ứng dụng chạy hoàn tất. Màn hình chờ LaunchScreen.storyboard cho phép thiết lập giao diện ứng dụng trong quãng thời gian chờ này. Hãy chọn LaunchScreen.storyboard trên Navigation Area, trong Utility Area chọn Attributes Inspector, trong mục background chọn màu đỏ (Hình 2.1.2.1). Chạy thử chương trình sẽ thấy trong thời gian chờ, màn hình màu đỏ sẽ hiện ra trong một khoảng thời gian ngắn.



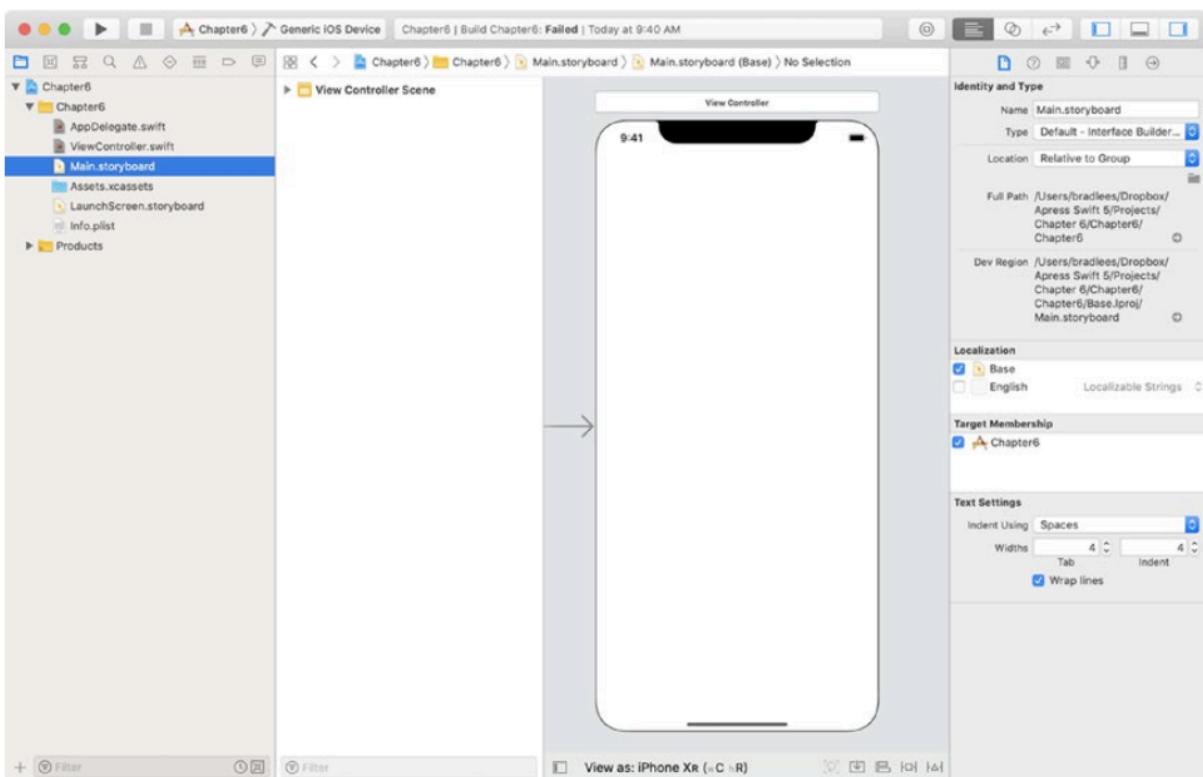
Hình 2.1.2.1 Cấu hình cho màn hình LaunchScreen trong ứng dụng iOS

2.1.3 Màn hình thiết kế giao diện Main.storyboard

Giao diện chính của màn hình hiện tại với các ứng dụng iOS hiện đại thường được thiết kế trong Main.storyboard với công cụ Interface Builder tích hợp sẵn trong Xcode. Trên Navigation Area lựa chọn Main.storyboard, trên vùng Editor giao diện để thiết kế cho

màn hình ứng dụng iOS hiện ra, cho phép lập trình viên xây dựng giao diện màn hình (Hình 2.1.3.1).

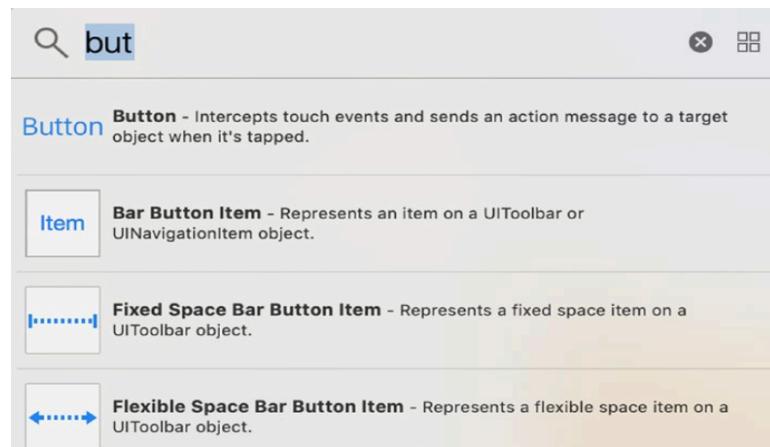
Hiện tại, Main.storyboard chỉ chứa giao diện thiết kế cho một màn hình (Các phần sau sẽ hướng dẫn cách tạo ra các ứng dụng nhiều màn hình và vấn đề di chuyển giữa chúng), đó là một khung hình trắng có hình dáng giống vẻ bên ngoài một chiếc iPhone và có một mũi tên trỏ vào cạnh bên trái. Khung hình trắng chính là khung giao diện cho lập trình viên tự thiết kế các phần tử bên trong của màn hình ứng dụng (Những gì hiện ra trong khung hình thì khi chạy chương trình cũng sẽ hiện ra trên màn hình iPhone, iPad tương ứng). Còn mũi tên được gọi là điểm vào của chương trình, nghĩa là với ứng dụng nhiều màn hình thì khung hình nào có mũi tên chỉ vào thì màn hình tương ứng với nó sẽ được thực hiện đầu tiên khi chạy chương trình (Giống như 2 lệnh cấu hình chạy chương trình trong Manifest của ứng dụng Android).



Hình 2.1.3.1 Màn hình thiết kế giao diện Main.storyboard

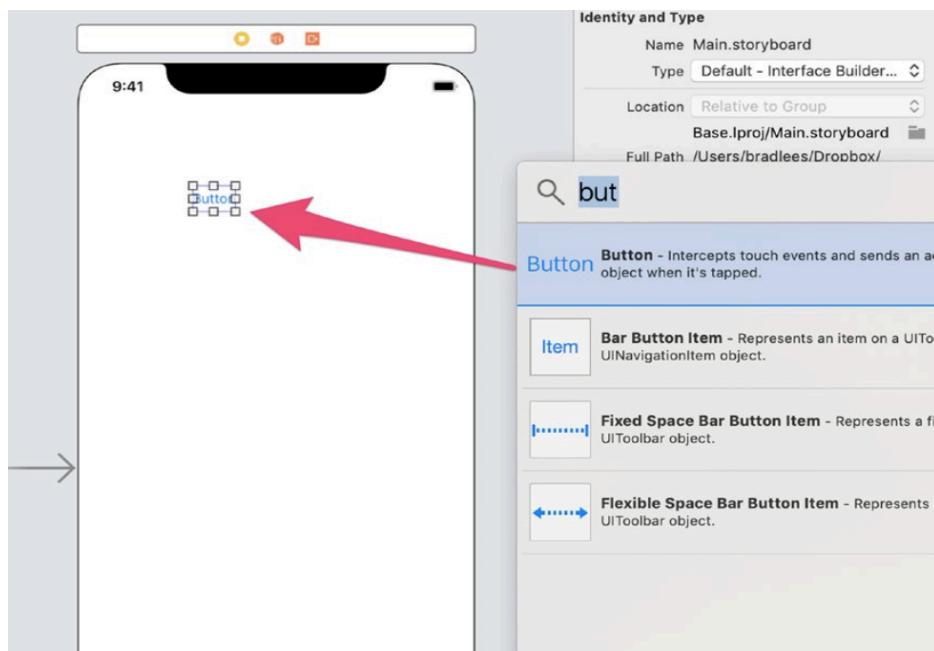
Thiết kế giao diện UI trong iOS tương đối dễ dàng, chủ yếu dựa trên thao tác kéo thả và đưa vào các ràng buộc cho các đối tượng giao diện. Xcode cung cấp sẵn cho người sử dụng một thư viện các đối tượng cho thiết kế giao diện để kéo vào trong các màn hình thiết kế trong Main.storyboard. Để có thể truy xuất vào thư viện này, nhấp chọn vào biểu tượng (trong các phiên bản Xcode cũ) hoặc biểu tượng (trong phiên bản

Xcode mới), khi đó giao diện cho phép lựa chọn và kéo thả các đối tượng trong thư viện (Object Library) sẽ xuất hiện (Hình 2.1.3.2).



Hình 2.1.3.2 Thư viện các đối tượng cho thiết kế giao diện (Object Library)

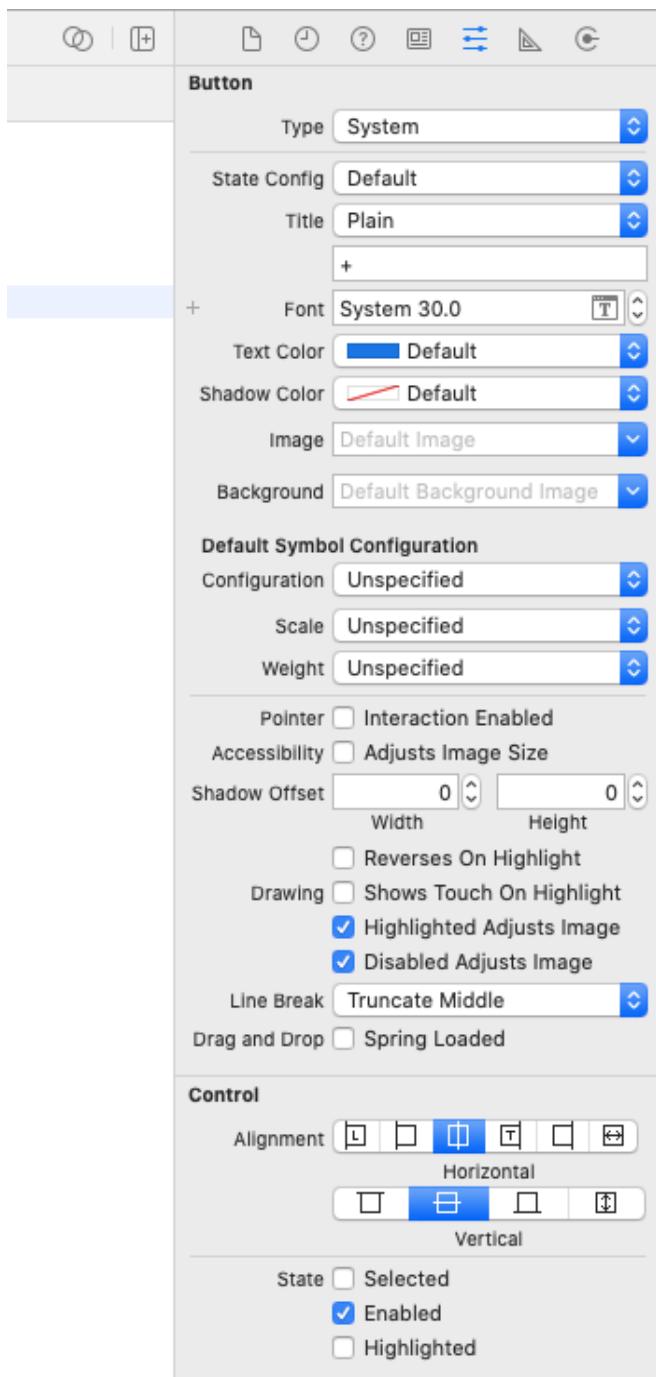
Trong màn hình Dialog này, lập trình viên hoàn toàn có thể sử dụng công cụ tìm kiếm gõ vào tên thư viện cần tìm để tìm nhanh đối tượng trong thư viện: Ví dụ gõ “but” thì các đối tượng loại Button như UIButton, Bar Button Item, Fixed Space Bar Button Item,... sẽ được lọc ra cho lập trình viên. Sau đó chỉ một thao tác kéo-thả đơn giản đối tượng tương ứng trong thư viện vào đối tượng màn hình trong Main.storyboard là chúng ta đã hoàn thành giai đoạn 1 của thiết kế giao diện (Hình 2.1.3.3).



Hình 2.1.3.3 Kéo – thả đối tượng từ thư viện vào màn hình thiết kế giao diện

Sau khi kéo đối tượng vào màn hình thiết kế giao diện, ta có thể thay đổi các thuộc tính của đối tượng tuỳ theo yêu cầu một cách dễ dàng dựa vào bảng thuộc tính của đối tượng (trong Utility Area):

- Nhấp đúp chuột lên trên đối tượng: Có thể thay đổi Title của đối tượng
- Lựa chọn đối tượng trên màn hình thiết kế giao diện => Mở Utility Area (Hình 2.1.3.4)



Lựa chọn biểu tượng để vào **Identity Inspector** của đối tượng (nhằm điều chỉnh các thuộc tính liên quan đến lớp đối tượng), hoặc biểu tượng để vào bảng Attributes Inspector của đối tượng (nhằm điều chỉnh các thuộc tính liên quan đến các tính chất của đối tượng như màu sắc, cẩn chỉnh lè, font chữ, màu chữ...), hoặc vào biểu tượng để vào Size Inspector của đối tượng (nhằm điều chỉnh các thuộc tính liên quan đến kích thước của đối tượng như dài, rộng, ...). Ngoài ra còn nhiều bảng thuộc tính khác sẽ được giới thiệu trong các phần tiếp theo. Ngoài ra, lập trình viên cũng có thể kéo và di chuyển đối tượng trong màn hình thiết kế giao diện đến những vị trí mong muốn, hoặc thay đổi trực tiếp kích thước các đối tượng bằng cách kéo các điểm giới hạn tại các cạnh hoặc góc của đối tượng như trong nhiều ngôn ngữ lập trình khác.

Hình 2.1.3.4 Bảng thuộc tính đối tượng trong thiết kế giao diện iOS

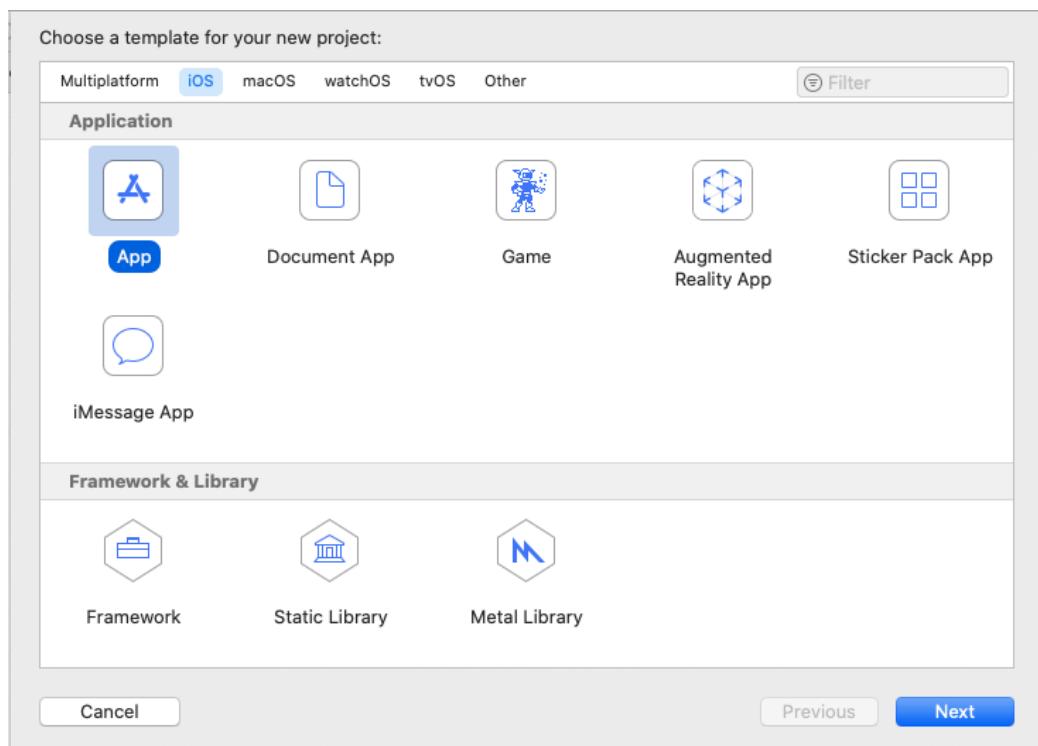


Hình 2.1.3.5 Ví dụ thay đổi thuộc tính đối tượng trực tiếp trên màn hình thiết kế

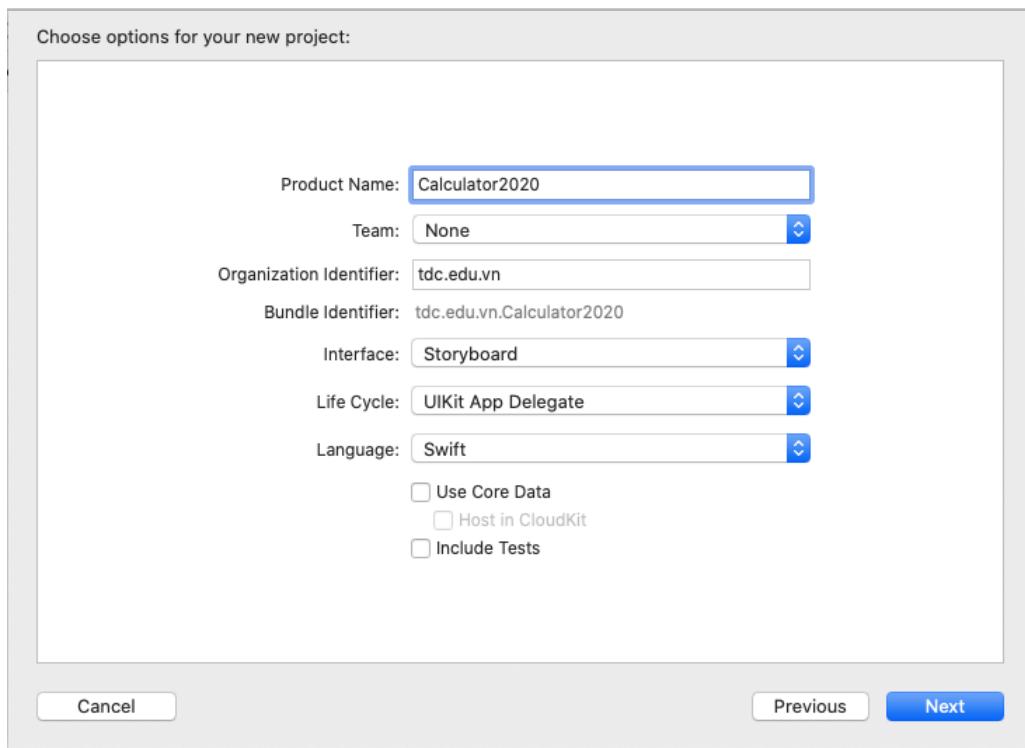
2.1.4 Case Study: Tạo ứng dụng Calculate

Trong toàn bộ giáo trình, chúng tôi sẽ hướng dẫn người học cách xây dựng hoàn chỉnh hai ứng dụng thực tiễn. Trong phần này, chúng ta sẽ cùng thực hiện ứng dụng thứ nhất, giai đoạn 1: Thiết kế giao diện cho ứng dụng máy tính bỏ túi **Calculator**.

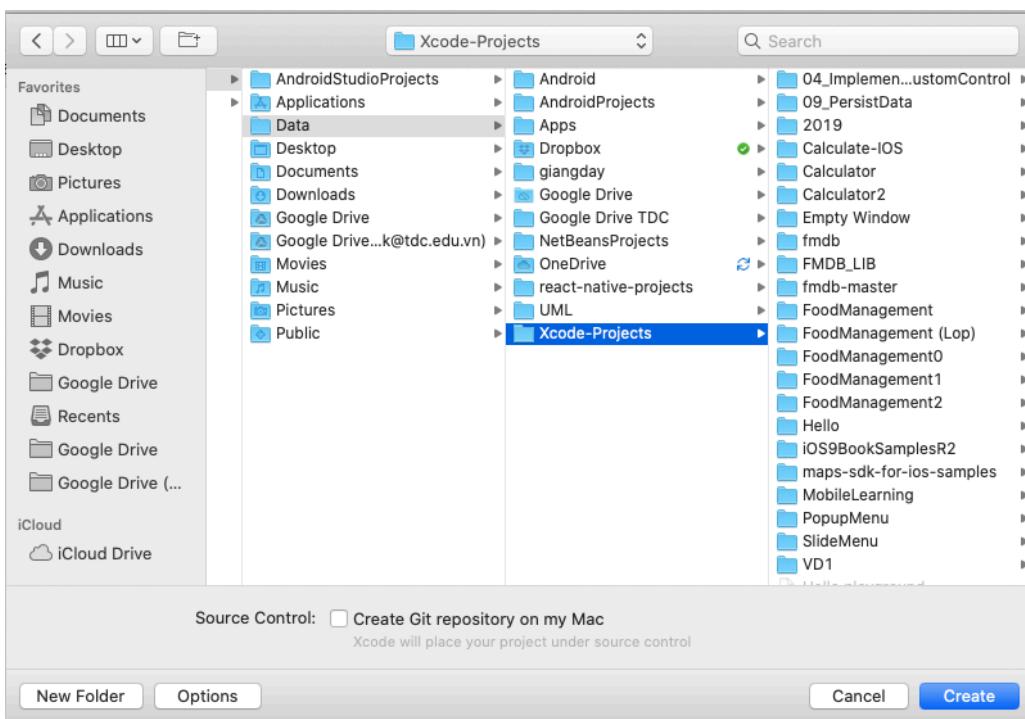
Bước 1: Tạo một iOS Project mới có tên Calculator2020



Hình 2.1.4.1 Tạo ứng dụng iOS Calculator2020 (a)



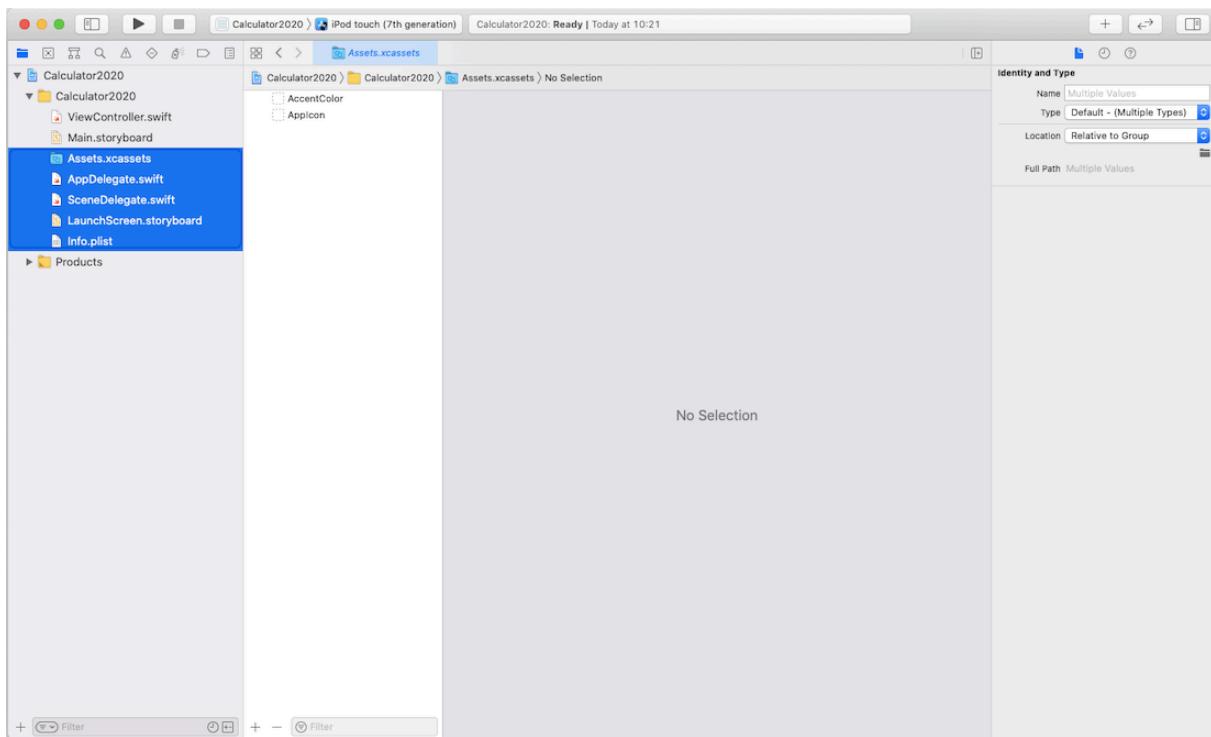
Hình 2.1.4.2 Tạo ứng dụng iOS Calculator2020 (b)



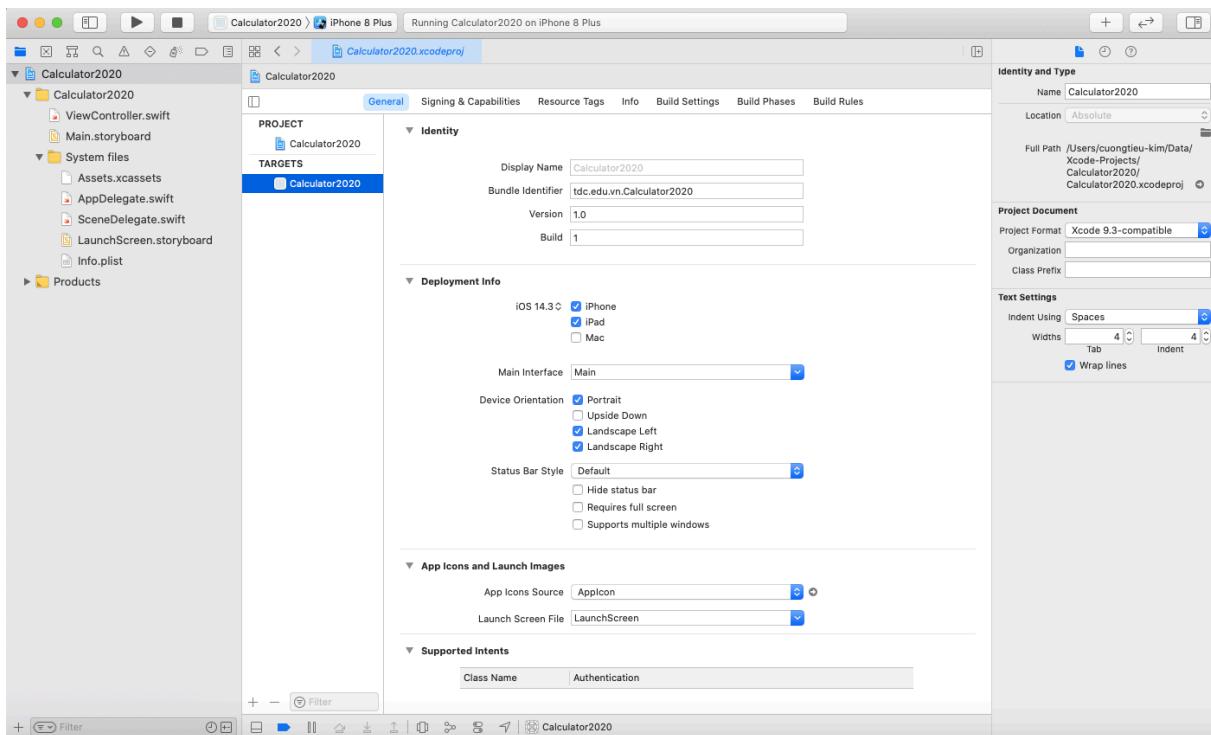
Hình 2.1.4.3 Tạo ứng dụng iOS Calculator2020 (c)

Bước 2: Thiết kế giao diện cho ứng dụng Calculator2020 giai đoạn 1

Trước tiên, trên vùng Navigation Area, gom nhóm các files AppDelegate.swift, ScenceDelegate.swift, LaunchScreen.storyboard, Info.plist và thư mục Assets vào một nhóm có tên là System files (Hình 2.1.4.4 và 2.1.4.5).



Hình 2.1.4.4 Gom nhóm các file hệ thống vào một nhóm để quản lý



Hình 2.1.4.5 Kết quả gom nhóm các file hệ thống trong System files

Mở Object Library, kéo thả và điều chỉnh thuộc tính cho các đối tượng như sau (thời điểm hiện tại, chưa quan tâm tới sự cân đối của màn hình giao diện):

- **Đối tượng UIButton:** Có Tiltle tương ứng với giá trị số của nó trên máy tính (Ví dụ 7), kích thước font chữ 30 kiểu in đậm đen, kích thước đối tượng 64x64, màu xám nhạt.

- **Đối tượng UILabel:** Thay đổi kích thước cho phù hợp độ rộng các Button, màu sắc Xanh da trời, kích thước font chữ 30 kiểu in đậm, căn lề bên phải, giá trị 0.

Kết quả bước đầu của phần thiết kế giao diện sẽ giống như **Hình 2.2.2.1** của bước tiếp theo trong mục 2.2.2.

Lựa chọn máy ảo loại iPhone 8 Plus và chạy thử để xem kết quả.

2.2 Xử lý sự kiện trên iOS

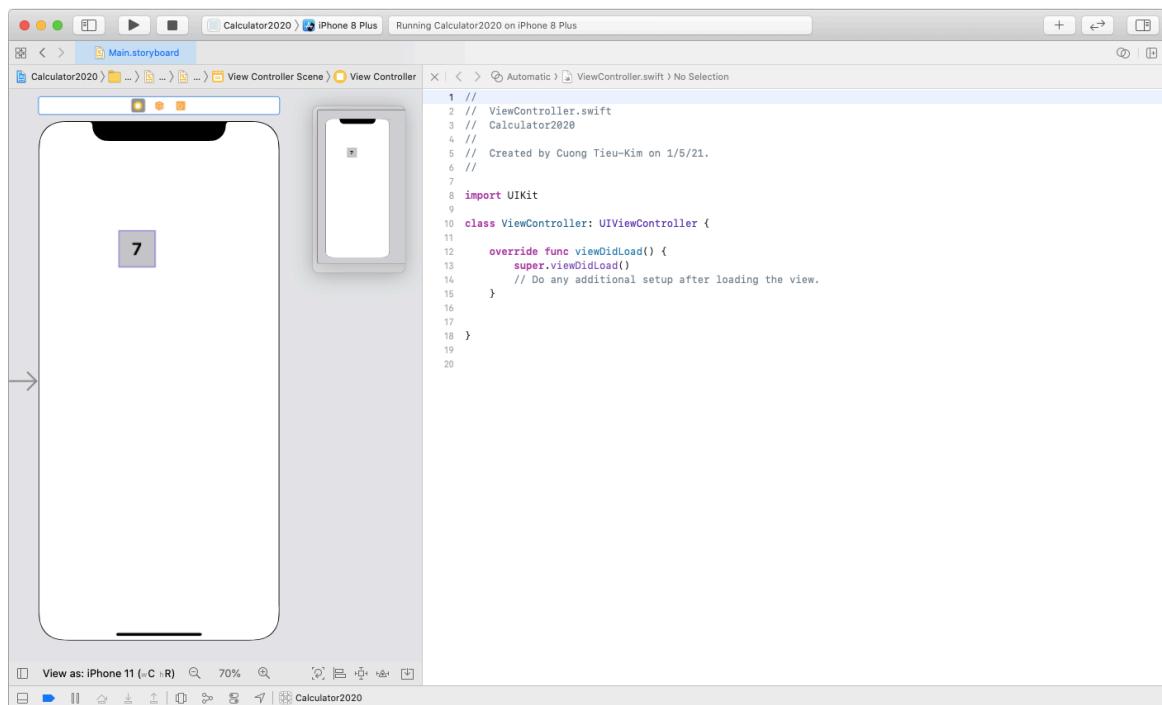
2.2.1 Kết nối các đối tượng với code

Cũng giống như viết các ứng dụng trên Android, cần lấy được các đối tượng trong layout trước khi viết code xử lý cho chúng. Trong các ứng dụng iOS, thì công việc này được làm trực tiếp thông qua việc “Kết nối” các đối tượng trong layout (trong storyboard) với các file xử lý sự kiện dạng *Controller.swift. Để có thể thực hiện việc kết nối, trước tiên chuyển về chế độ kết nối trước:

- Lựa chọn Main.storyboard

- Trên thanh công cụ chọn Editor => Assistant

Màn hình liên kết code xuất hiện với giao diện storyboard bên trái và code editor cho file *Controller.swift tương ứng với nó bên phải (Hình 2.2.1.1).



Hình 2.2.1.1 Màn hình liên kết code trong lập trình iOS

Trên giao diện storyboard, lựa chọn đối tượng sẽ liên kết code (Button 7), bấm và giữ phím Ctrl trên bàn phím cùng lúc Click và kéo đối tượng từ storyboard sang màn hình code Editor, thả vào vị trí mong muốn (nếu là tham chiếu đến đối tượng thì thả trong khu Properties của lớp, nếu là hàm thực hiện hành vi của đối tượng thì thả trong phần định nghĩa các methods của lớp). Khi đó một Dialog sẽ xuất hiện (Hình 2.2.1.2).



Hình 2.2.1.2 Thiết lập tham số cho việc liên kết code với đối tượng trong layout

- Ở mục Connection: Lựa chọn là Action (liên kết code dạng thực hiện hành vi cho đối tượng trong layout).
- Ở mục Name: Gõ tên của hàm sẽ được gọi mỗi khi đối tượng này được tác động.
- Ở mục Type: Lựa chọn kiểu của đối tượng trong layout là UIButton.
- Ở mục Event: Lựa chọn loại sự kiện tác động lên đối tượng. Ở đây ta lựa chọn **Touch Up Inside** (nghĩa là mỗi khi tap lên trên đối tượng sau đó nhả tay cũng trên đối tượng đó chứ không di chuyển ra ngoài mới nhả... thì hàm này sẽ được gọi).

Nhấp chọn Connect để thiết lập liên kết code và ta được kết quả mong muốn trong phần Code Editor (Hình 2.2.1.3).

```

7
8 import UIKit
9
10 class ViewController: UIViewController {
11
12     override func viewDidLoad() {
13         super.viewDidLoad()
14         // Do any additional setup after loading the view.
15     }
16
17     @IBAction func buttonPress(_ sender: UIButton) {
18
19     }
20
21 }
```

Hình 2.2.1.3 Kết quả liên kết code dạng hành vi (Action) của đối tượng

Nếu ở mục Connection chúng ta lựa chọn **Outlet** thay vì **Action** thì liên kết code dạng tham chiếu sẽ được tạo. Khi đó, ta có thể dùng biến tham chiếu này để truy xuất các đối tượng trên layout như một biến tham chiếu đến đối tượng bất kỳ.

2.2.2 Cách viết hàm trong iOS với ngôn ngữ Swift

Hàm buttonPress() ở trên được tạo ra sau khi liên kết code dạng hành vi với button 7 có cấu trúc tương tự những hàm được thiết kế trong ngôn ngữ Swift (Chương 1). Tuy nhiên, có một điểm khác biệt đó là chỉ báo biên dịch `@IBAction` đặt trước từ khoá func của định nghĩa hàm, đó là chỉ báo chỉ ra rằng hàm này có liên kết code với giao diện storyboard của ứng dụng (IB viết tắt của Interface Builder). Còn những hàm, phương thức khác đều được viết theo đúng quy tắc, cú pháp của ngôn ngữ Swift đã được học trong chương 1.

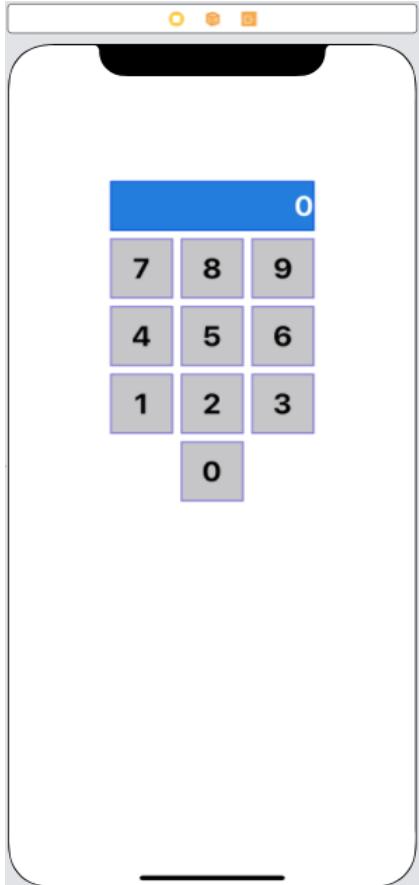
Trước tiên chúng ta muốn hàm thực hiện thao tác đơn giản là hiện giá trị của số tương ứng với nút được chạm vào trên màn hình giao diện (số 7) lên màn hình consol khi chạy chương trình. Để làm việc đó, trong thân hàm ta bổ sung thêm 2 dòng lệnh:

```
@IBAction func buttonPress(_ sender: UIButton) {
    let digit = sender.currentTitle!
    print("The button \(digit) duoc nhan!")
}
```

Ở đây, mỗi khi chạy chương trình và chạm vào nút button 7 trên giao diện màn hình máy ảo iPhone, thì hàm kết nối code tương ứng buttonPress(sender:) trong mục 2.2.1 sẽ được gọi và biến **sender** truyền vào hàm sẽ chính là đối tượng button 7 được chạm vào. Khi đó biến `digit = sender.currentTiltle!` sẽ chính là giá trị số 7 dưới dạng String (Vì nó là biến Optional nên cần phải unWrap nó bằng !). Kết quả, trên màn hình consol sẽ hiện lên dòng: “The button 7 duoc nhan!”.

Thực hiện copy nút button 7 và dán vào giao diện và sửa đổi tiêu đề của chúng để có màn hình giao diện như dưới đây (Hình 2.2.2.1).

Trong iOS, các đối tượng trong layout khi được copy sẽ copy cả những thuộc tính liên kết code tương ứng của đối tượng đó. Trong ví dụ này, tất cả 10 button trên giao diện giờ đây đều tham chiếu đến cùng hàm buttonPress(sender:) ở trên. Hãy chạy chương trình với Simulator là iPhone 8 Plus, tap chọn các button trên giao diện và cho kết luận!



2.2.3 Hoàn thiện ứng dụng Calculate giai đoạn 1

Giai đoạn 1 của ứng dụng cần xử lý nhiều thao tác như mỗi khi tap trên một đối tượng button cụ thể thì thay vì hiện kết quả trên màn hình consol của Xcode thì giá trị số được chạm sẽ hiện trên màn hình kết quả của máy tính; xử lý các trường hợp số không đầu tiên, xử lý các phím chức năng...

Trước tiên chúng ta đưa kết quả của các phím số lên màn hình của ứng dụng Calculator: Thực hiện kết nối màn hình Calculator UILabel với code theo dạng tham chiếu và đặt tên là `calDisplay`.

Bước 1: Hãy sửa hàm `buttonPress(sender:)` sao cho mỗi khi một button được chạm vào thì giá trị số tương ứng được ghi dồn vào màn hình của Calculator như sau:

Hình 2.2.2.1 Kết quả điều chỉnh giao diện giai đoạn 1

```
//MARK: Calculator behaviors
@IBAction func buttonPress(_ sender: UIButton) {
    let digit = sender.currentTitle!
    let currentCalDisplay = calDisplay.text!
    calDisplay.text = currentCalDisplay + digit
}
```

Biến `currentCalDisplay` sẽ lưu giá trị hiện tại trên màn hình Calculator trước khi được nối với số vừa được chạm vào (`digit`). Chạy chương trình và ta thu được kết quả như hình 2.2.2.2. Tuy nhiên, rõ ràng đó chưa phải kết quả mong muốn vì số 0 ban đầu vẫn được nối vào trước của số được chọn trên màn hình máy tính, điều này không đúng với thực tế. Ngoài ra cũng cần xét thêm trường hợp nếu nút đầu tiên được chạm cũng là nút số 0 thì cũng không được phép gắn nó lên màn hình Calculator (trừ khi nó được chạm sau khi trên màn hình đã có các số khác 0). Hãy sửa lại hàm `buttonPress(sender:)` như sau để có kết quả mong muốn:

```
//MARK: Properties
@IBOutlet weak var calDisplay: UILabel!
var isTyping = false

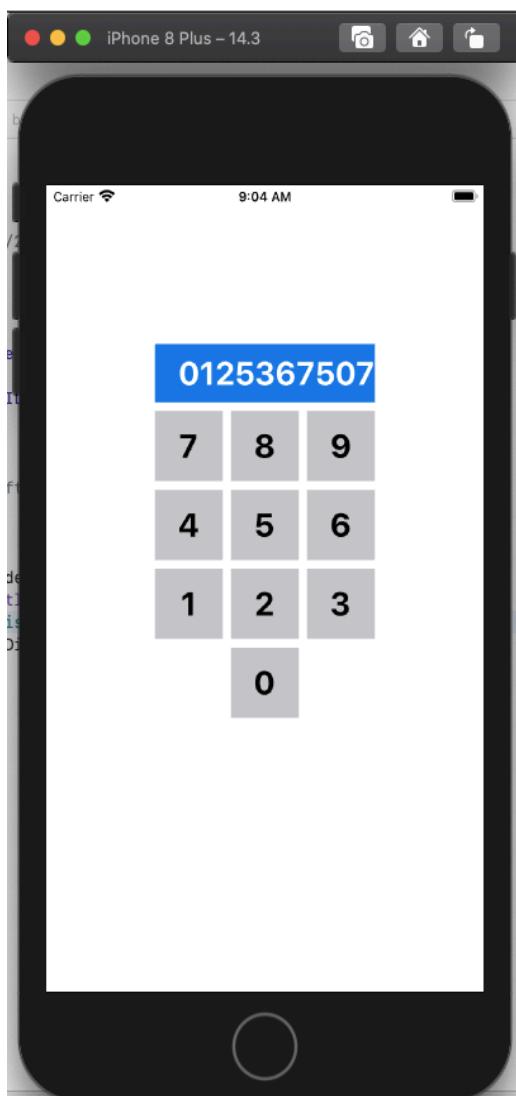
//MARK: Calculator behaviors
@IBAction func buttonPress(_ sender: UIButton) {
```

```

let digit = sender.currentTitle!
if isTyping {
    let currentCalDisplay = calDisplay.text!
    calDisplay.text = currentCalDisplay + digit
}
else {
    if digit != "0" {
        calDisplay.text = digit
        isTyping = true
    }
}

```

Hãy chạy chương trình, tap vào các phím số với mọi khả năng có thể có và cho kết luận!



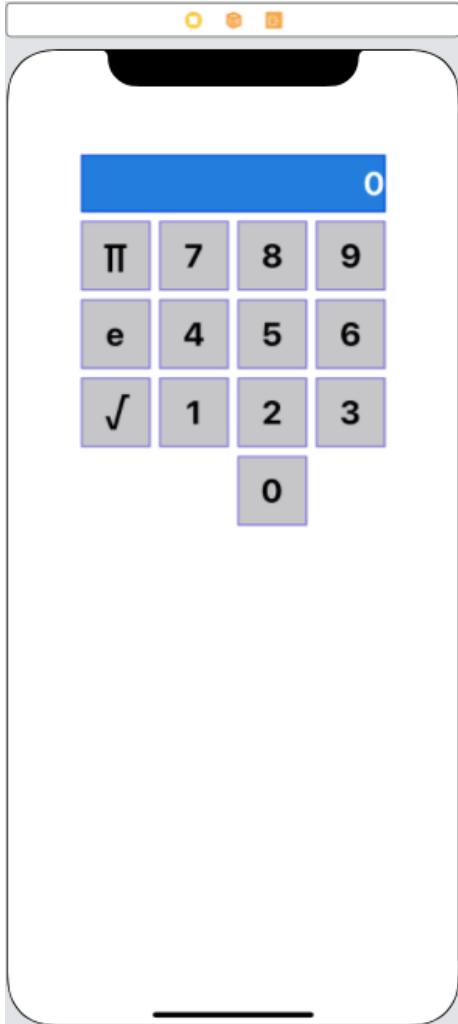
Bước 2: Thêm phím chức năng và kết nối code dạng hành vi cho các chức năng này với hàm có tên là calFunctions(sender:) với cùng sự kiện là Touch Up Inside như các phím số trước đây để có kết quả như Hình 2.2.2.3. Lưu ý: Để có thể đưa giá trị các phím chức năng như π , ... vào các button cần truy xuất vào bảng ký tự đặc biệt trong Xcode để lấy (Hình 2.2.2.4). Trong bước này, tạm thời chúng ta sẽ thêm 3 phím chức năng là phím lấy giá trị Pi, e và lấy căn bậc hai.

Mỗi khi một phím chức năng được chạm thì chức năng tương ứng sẽ được thực hiện. Trong trường hợp của chúng ta thì:

- Chức năng π : Màn hình Calculator sẽ hiển thị giá trị của số Pi.
- Chức năng e: Màn hình Calculator sẽ hiển thị giá trị của số e.

Hình 2.2.2.2 Kết quả điều chỉnh chương trình

- Chức năng $\sqrt{}$: Màn hình Calculator sẽ hiển thị kết quả phép tính căn bậc hai của giá trị hiện tại trên màn hình của máy tính. Rõ ràng ở đây cần xử lý khéo léo giữa phím chức năng và phím số, nếu không ứng dụng sẽ hoạt động hỗn loạn.



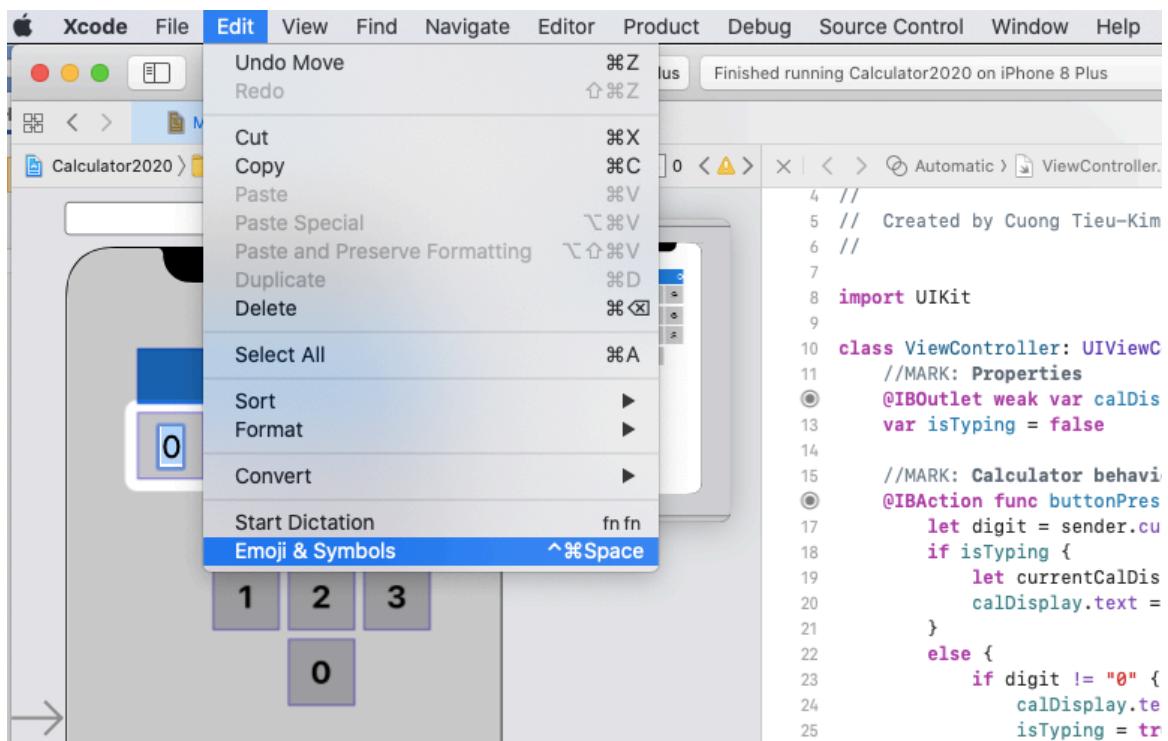
Điều chỉnh code trong hàm mới kết nối như sau:

```
@IBAction func calFunctions(_ sender: UIButton) {
    if let mathSymbol = sender.currentTitle {
        isTyping = false
        switch mathSymbol {
        case "π": calDisplay.text = String(Double.pi)
        case "e": calDisplay.text = String(M_E)
        case "√": if let operand = Double(calDisplay.text!) {
                    calDisplay.text = String(sqrt(operand))
                }
        default: print("This is not a function")
        }
    }
}
```

Lưu ý:

- Mỗi khi hàm chức năng được gọi thì cần xoá lại trạng thái isTyping về false để đảm bảo sau khi gọi phím chức năng thì các phím số vẫn hoạt động như bình thường.
- unWrap code an toàn cho trường hợp tính căn bậc hai, để tránh trường hợp không lấy được giá trị.

Hình 2.2.2.3 Bổ sung phím chức năng cho máy tính



Hình 2.2.2.4 Lấy các ký tự đặc biệt trong Xcode

Computed Variable và việc hoàn thiện chương trình Calculator giai đoạn 1:

Trong ứng dụng trên, mỗi khi sử dụng phím chức năng và phím số, rõ ràng luôn cần sự chuyển đổi giữa String và giá trị số tương ứng của màn hình Calculator. Điều này có thể giải quyết dễ dàng trong các ứng dụng iOS với kiểu biến tính toán (Computed Variable) trong Swift. Biến tính toán cho phép định nghĩa các hàm get và set để thực hiện các thao tác chuyển đổi, tính toán tương ứng trước khi sử dụng và khi sử dụng giống như ta đưa/lấy trực tiếp giá trị Double vào/ra màn hình calDisplay của máy tính. Sửa lại code trong chương trình như sau:

```
var disPlayValue: Double {
    get {
        if let value = Double(calDisplay.text!) {
            return value
        }
        else {
            return 0
        }
    }
    set {
        calDisplay.text = String(newValue)
    }
}
@IBAction func calFunctions(_ sender: UIButton) {
    if let mathSymbol = sender.currentTitle {
        isTyping = false
        switch mathSymbol {
        case "π": disPlayValue = Double.pi
        case "e": disPlayValue = M_E
        case "√":
            disPlayValue = sqrt(disPlayValue)
        default:
            print("This is not a function")
        }
    }
}
```

2.3 Tô chúc code theo mô hình MVC

2.3.1 Phân tích ứng dụng theo mô hình MVC

Trong chương 1 chúng ta đã hiểu về mô hình MVC. Trong phần này, chúng ta sẽ ứng dụng mô hình MVC vào phân tích, thiết kế và triển khai ứng dụng Calculator để hoàn thiện cho ứng dụng này.

Tô chúc hoạt động: Nếu cần bổ sung thêm nhiều tính năng khác nhau cho máy tính như các hàm tính toán (\pm , \sin , \cos , ...), các phép toán (+, -, x , \div , ...), thậm chí các hàm số nào đó... thì ta cần điều chỉnh như thế nào? Độ phức tạp của việc điều chỉnh? Đề xuất

giải pháp để hạn chế nhiều nhất việc điều chỉnh nếu có thể? Gợi ý cho sinh viên thực hiện theo hướng tiếp cận của mô hình MVC:

Cần tách phần chức năng của máy tính thành một lớp riêng. Mọi hoạt động tính toán đều thông qua lớp này (vai trò M – Model). Mọi tương tác của người dùng (vai trò V - View) đều được ghi nhận và chuyển tới khu vực điều khiển (vai trò C – Controller), từ đây yêu cầu sẽ được chuyển tới M để yêu cầu việc thực hiện tính toán và lấy lại kết quả (với các chức năng) hoặc hiển thị lại lên giao diện người dùng (với các phím số). Sau khi nhận lại kết quả từ M, C sẽ chuyển lại kết quả cho V để hiển thị kết quả. Vấn đề ở đây là cần thiết kế các thuộc tính? Phương thức? như thế nào để đảm bảo việc trao đổi trên diễn ra một cách tốt đẹp? bao hàm mọi tình huống có thể có?

2.3.2 Xây dựng ứng dụng theo mô hình MVC

Bước 1: Tách lớp Model cho ứng dụng Calculator.

Tạo group mới có tên **models** trong đó tạo một file mới với lớp mới có tên **CalculatorBrain** (Chọn loại Swift file). Mọi hoạt động “nghiệp vụ” của máy tính sẽ được xây dựng, cập nhật và mở rộng ở lớp này (vai trò M).

Nội dung lớp CalculatorBrain sẽ như sau:

```
import Foundation
class CalculatorBrain {
    // To store the template calculated value
    private var accumulator: Double?

    // To get the current Value of calculator's Screen
    func setOperand(_ operand: Double) {
        accumulator = operand
    }
    // To perform the requested Calculation
    func requestCalculate(mathSymbol: String) {
        switch mathSymbol {
        case "π": accumulator = Double.pi
        case "e": accumulator = M_E
        case "√":
            if let operand = accumulator {
                accumulator = sqrt(operand)
            }
        default:
            print("This is not a function")
        }
    }
    // Return the result of the Calculation
    var result: Double? {
        get {
```

```
        return accumulator  
    }  
}
```

Bước 2: Điều chỉnh lại lớp ViewController cho phù hợp với vai trò của một Controller:

Nội dung điều chỉnh sẽ như sau (Các phần khác không có sự thay đổi):

```
var calBrain = CalculatorBrain()

@IBAction func calFunctions(_ sender: UIButton) {
    // Transform calculator's screen value to the Brain
    if isTyping {
        calBrain.setOperand(displayValue)
        isTyping = false
    }
    // Request the Brain to perform the Calculation
    if let mathOperation = sender.currentTitle {
        calBrain.requestCalculate(mathSymbol: mathOperation)
    }
    // Get the result and set to the Screen
    if let result = calBrain.result {
        displayValue = result
    }
}
```

Bài tập: Hãy thực hiện chương trình và cho nhận xét!

2.3.3 Mở rộng và hoàn thiện ứng dụng Calculate

Trong phần này chúng ta thực hiện mở rộng thêm các chức năng khác cho máy tính Calculator như: các hàm 1 ngôi (\sin , \cos , \pm), các hàm hai ngôi ($+$, $-$, \times , \div). Việc cập nhật, bổ sung, điều chỉnh luôn được thực hiện trong lớp `CalculatorBrain` (và không thay đổi trong các lớp khác, ngoại trừ những điều chỉnh thêm về giao diện - V).

Với những chức năng mới, rõ ràng code cũ không đáp ứng được vì với một biến truyền vào mathSymbol hàm **requestCalculate()** không thể xác định được đâu là chức năng chỉ hiện thị ra các hằng số (π , e), đâu là chức năng để tính toán cho các hàm một tham số (\sin , \cos , \pm) và đâu là chức năng thực hiện cho các hàm hai tham số (+, -, x , \div), và với các hàm hai tham số thì cần phân biệt khi nào truyền tham số (khi nhập vào toán hạng đầu tiên) và khi nào thì phép toán được yêu cầu thực hiện (phải có phím = để yêu cầu thực hiện phép toán hai ngôi). Nếu thực hiện theo cách thông thường, thì cấu trúc chương trình rất phức tạp, đòi hỏi nhiều điều kiện lồng nhau. Tuy nhiên, với Swift, ta có thể sử dụng nhiều cấu trúc dữ liệu thay thế để chương trình sáng sủa hơn.

Trước tiên, để phân biệt các loại phép toán (hằng số, phép toán một ngôi, phép toán hai ngôi, dấu bằng) chúng ta sẽ sử dụng cấu trúc dữ liệu enum (như trong Java) và để xử lý từng loại thành phần trong enum ta sử dụng cấu trúc Dictionary. Để dễ hiểu, trước tiên chúng ta thay thế cấu trúc switch bằng cấu trúc dữ liệu Dictionary cho hai chức năng Pi và số e ta có thể dễ dàng thực hiện như sau:

```
// Define the data structure of math operations
private var operations: Dictionary<String, Double> = [
    "π": Double.pi,
    "e": M_E
]
// To perform the requested Calculation
func requestCalculate(mathSymbol: String) {
    if let constant = operations[mathSymbol] {
        accumulator = constant
    }
}
```

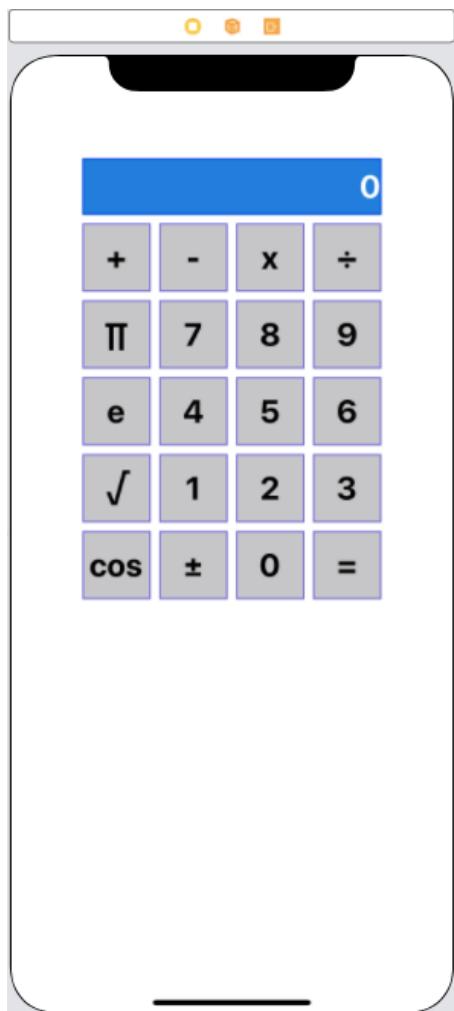
Tuy nhiên, khi chức năng là một phép toán 1 ngôi (sqrt, sin, cos...) thì vẫn đề chưa thể được xử lý vì chúng là một hàm chứ không phải một giá trị Double. Ta cần định nghĩa thêm một cấu trúc dữ liệu enum để phân loại các loại chức năng như sau:

```
// Define the type of Operation
private enum Operation {
    case constant(Double)
    case unaryOperation((Double)->Double)
    case binaryOperation
    case equal
}
// Define the data structure of math operations
private var operations: Dictionary<String, Operation> = [
    "π": Operation.constant(Double.pi),
    "e": Operation.constant(M_E),
    "√": Operation.unaryOperation(sqrt)
]
// To perform the requested Calculation
func requestCalculate(mathSymbol: String) {
    if let mathOperation = operations[mathSymbol] {
        switch mathOperation {
            case .constant(let constantValue):
                accumulator = constantValue
            case .unaryOperation(let mathFunction):
                if let operand = accumulator {
                    accumulator = mathFunction(operand)
                }
            default:
                break
        }
    }
}
```

Trong đó, tạm thời .constant của enum sẽ nhận tham số truyền vào là một giá trị Double (có thể là Pi hoặc e) và .unaryOperation sẽ nhận tham số truyền vào là một biến hàm cho các phép toán một ngôi (Double) -> Double. Trong hàm requestCalculate() cần phải lấy ra dạng phép toán tương ứng (constant, phép toán 1 ngôi, phép toán 2 ngôi hay dấu =) với mỗi ký tự toán học được truyền từ Controller sang (“Π”, “e”, “√”,...). Điều này dễ dàng thực hiện thông qua phép truy xuất từ cấu trúc Dictionary. Tuy nhiên, để xử lý từng trường hợp cụ thể của loại phép toán, ta lại phải dùng cấu trúc switch.

Bài tập: Thực hiện chương trình và cho nhận xét!

Tiếp theo, ta mở rộng thêm chức năng của máy tính Calculator: với sin, cos, ± cho phép toán một ngôi và +, -, ×, ÷ cho phép toán hai ngôi. Giao diện sẽ như hình 2.3.3.1.



Với phép toán 1 ngôi “√” và “cos” ta có thể sử dụng hàm có sẵn trong thư viện bằng cách truyền tên hàm tương ứng trong cấu trúc Dictionary ở trên. Tuy nhiên, với những phép toán còn lại, ta cần định nghĩa hàm tương ứng cho chúng:

```
import Foundation
func changeSign(_ operand: Double) ->
Double {
    return -operand
}
func mul(_ a: Double, b: Double) -> Double
{
    return a * b
}
func div(_ a: Double, _ b: Double) ->
Double {
    return a/b
}
func add(_ a: Double, _ b: Double) ->
Double {
    return a + b
}
func sub(_ a: Double, _ b: Double) ->
Double {
    return a - b
}
```

Hình 2.3.3.1 Giao diện chức năng bổ sung

Bài tập: Hãy điều chỉnh lớp **CalculatorBrain** sao cho có thể thực hiện cho tất cả các phép toán một ngôi (cos, ±) trên giao diện máy tính mới cập nhật!

Tổ chức hoạt động: Tìm giải pháp thực hiện cho phép toán hai ngôi!

Gợi ý:

Với phép toán hai ngôi, công việc sẽ phức tạp hơn nhiều do chúng cần phải thực hiện qua hai bước đó là nhập toán hạng thứ nhất và nhập phép toán sau đó nhập tiếp toán hạng thứ hai và phím “=” thì Calculator sẽ thực hiện và cho ra kết quả. Do lớp CalculatorBrain chỉ chứa một biến để lưu giá trị hiện tại trên màn hình của máy tính (biến accumulator) nên khi người dùng nhập toán hạng thứ 2 của phép toán hai ngôi, thì toán hạng thứ nhất sẽ bị mất đi. Do vậy trong trường hợp này ta cần một cấu trúc dữ liệu để lưu lại trạng thái của máy tính trước đó (toán hạng 1, tên của hàm được gọi) đồng thời có một phương thức để thực hiện phép toán này với toán hạng thứ 2.

Với những phân tích đó, rõ ràng ở đây ta cần dùng một struct hoặc một class để thực hiện. Trong trường hợp này ta dùng struct:

```
//Struct to store information of binaryOperations
private struct PendingBinaryOperation {
    // Store the function
    let theBinaryFunction: (Double, Double) -> Double
    // Store the first operand
    let theFirstOperand: Double
    // Function to perform the function with the second operand
    func performCalculate(secondOperand: Double) -> Double {
        return theBinaryFunction(theFirstOperand, secondOperand)
    }
}

private var storeForBinaryFunctions: PendingBinaryOperation?
```

Và hàm requestCalculate được thay đổi như sau:

```
// To perform the requested Calculation
func requestCalculate(mathSymbol: String) {
    if let mathOperation = operations[mathSymbol] {
        switch mathOperation {
            case .constant(let constantValue):
                accumulator = constantValue
            case .unaryOperation(let mathFunction):
                if let operand = accumulator {
                    accumulator = mathFunction(operand)
                }
            case .binaryOperation(let mathFunction):
                if let firstOperand = accumulator {
                    storeForBinaryFunctions =
                        PendingBinaryOperation(theBinaryFunction:
                            mathFunction, theFirstOperand: firstOperand)
                }
            case .equal:
```

```
        if storeForBinaryFunctions != nil && accumulator != nil
    {
        accumulator =
            storeForBinaryFunctions?.performCalculate(
                secondOperand: accumulator!)
        storeForBinaryFunctions = nil
    }
}
```

Bài tập: Hãy điều chỉnh cấu trúc enum và cấu trúc Dictionary để hoàn thiện chương trình và máy tính hoạt động bình thường với tất cả các chức năng trên giao diện.

2.4 Autolayout trong iOS

2.4.1 Vấn đề giao diện trong ứng dụng Calculate

Hãy xoay màn hình máy ảo, dễ nhận thấy giao diện ứng dụng không còn như mong muốn (Hình 2.4.1.1), hoặc đôi khi giao diện bị vỡ không thể nhìn (nếu ứng dụng sử dụng các loại hình ảnh với kích thước khác nhau...).



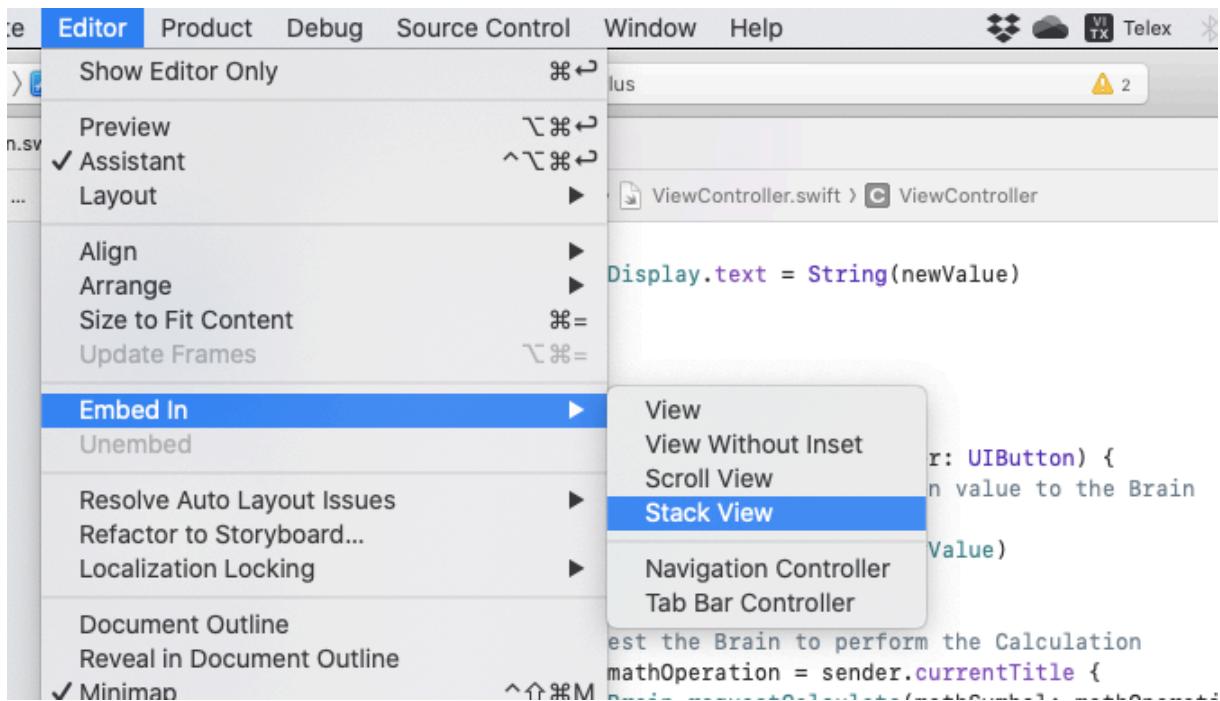
Hình 2.4.1.1 Vấn đề thay đổi hướng nhìn của các giao diện iOS

2.4.2 Cải tiến giao diện cho ứng dụng Calculate

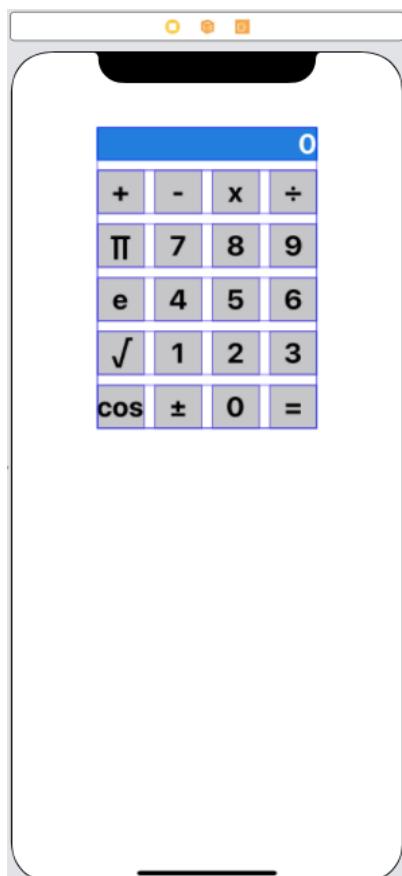
Để giải quyết vấn đề xảy ra cho các ứng dụng iOS như đề cập trong mục 2.4.1, chúng ta cần thiết kế lại chúng theo hướng tiếp cận Autolayout (Các giao diện sẽ tự thay đổi hình dáng cho phù hợp với các kiểu màn hình khác nhau).

Bước 1: Đưa các đối tượng Button vào trong các StackView, theo từng hàng

Lựa chọn các buttons trên từng hàng => Editor => Embed In => Stack View



Hình 2.4.2.1 Nhúng các đối tượng trong Stack View

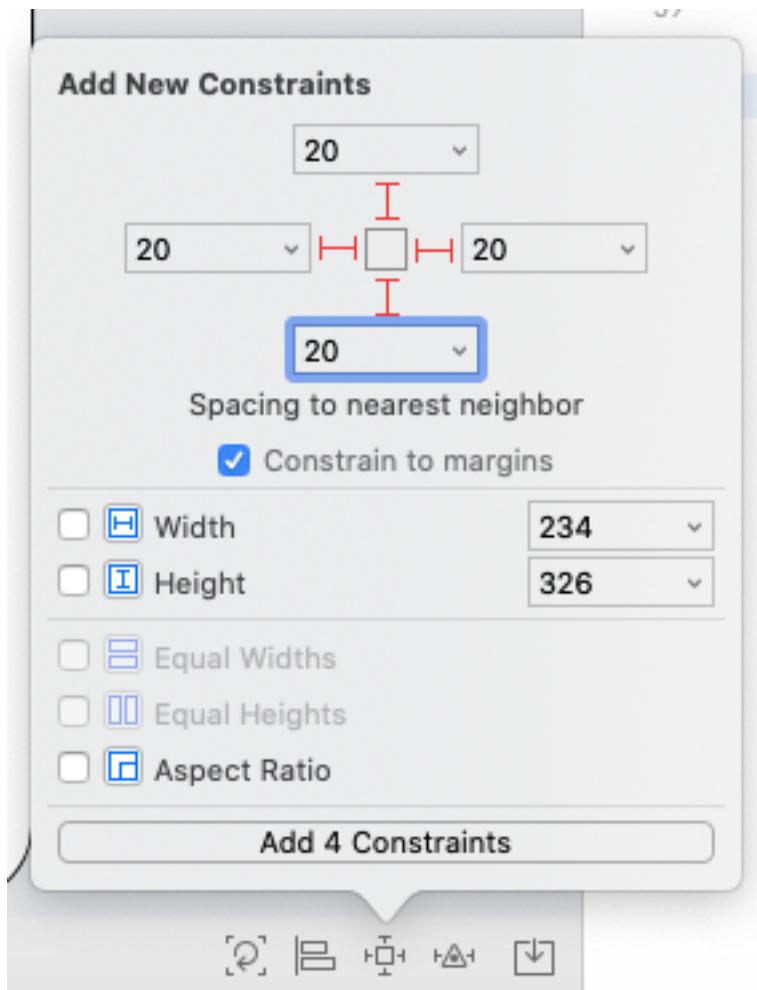


Trong Attributes Inspector thay đổi thuộc tính: Spacing = 10, Distribution = Fill Equally. Thực hiện cho tất cả các hàng. Tiếp theo, lại lựa chọn tất cả các hàng và nhúng chúng vào trong một Stack View khác. Trong Attributes Inspector thay đổi thuộc tính: Spacing = 10, Distribution = Fill Equally và Allignment = Fill. Tiếp theo, lại nhúng toàn bộ các Button với màn hình của máy tính trong một Stack View khác. Trong Attributes Inspector thay đổi thuộc tính: Spacing = 10, Distribution = Fill và Allignment = Fill. Kết quả thu được như hình 2.4.2.2.

Bước 2: Thêm các ràng buộc để đảm bảo các đối tượng sẽ tự điều chỉnh kích thước cho phù hợp mỗi khi thay đổi độ phân giải màn hình hoặc hướng của màn hình. Để ý rằng, giờ đây các đối tượng được coi như là 1 StackView.

Hình 2.4.2.2 Kết quả sau khi nhúng các đối tượng trong Stack View

Nên nhiệm vụ của chúng ta là thêm các ràng buộc của StackView này với 4 cạnh của màn hình điện thoại. Lựa chọn Stack View cuối cùng rồi nhấp chọn biểu tượng bên dưới màn hình, màn hình thêm ràng buộc cho layout xuất hiện (Hình 2.4.2.3).



Hình 2.4.2.3 Thêm ràng buộc cho Autolayout

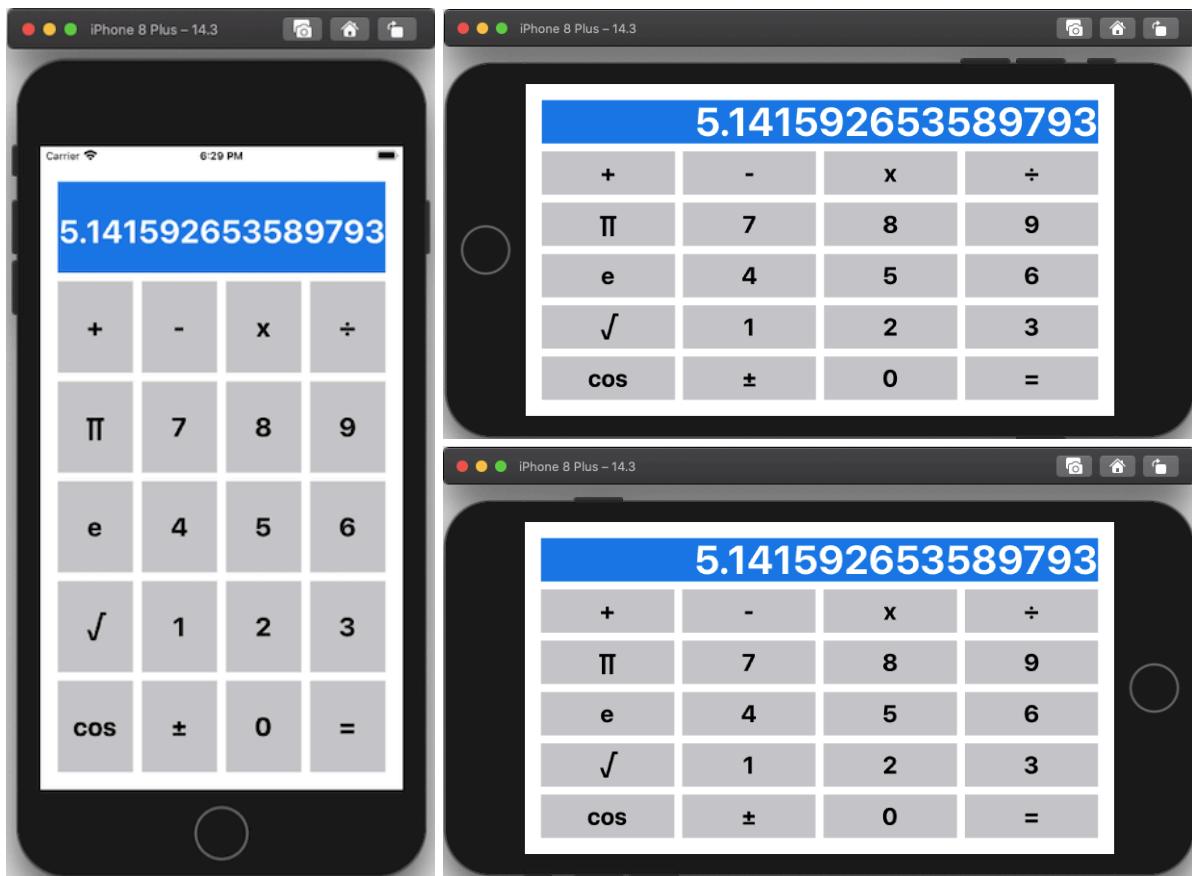
Sửa các giá trị cho lề trái, phải, trên dưới đều là 20 rồi nhấn “**Add 4 Constraints**” để hoàn thành cho Autolayout.

Khi đó giao diện sẽ tự động co dãn sao cho lề trái, lề phải, lề trên và lề dưới của nó đều có giá trị là 20.

Lưu ý: Để xem nhanh cấu trúc cây của các View trong màn hình, nhấn giữ Shift + Phải chuột trên đối tượng view muốn xem.

Chạy chương trình và xoay màn hình ta có các kết quả như hình 2.4.2.4.

Bài tập: Hãy điều chỉnh giao diện sao cho chiều cao của các button và chiều cao của màn hình máy tính là giống nhau!



Hình 2.4.2.4 Autolayout với các chiều khác nhau của màn hình iPhone

2.5 Case Study: Thiết kế ứng dụng Quản lý món ăn

Quản lý món ăn là một ứng dụng tương đối phức tạp đối với người mới học về lập trình iOS, tuy nhiên, giáo trình sẽ trình bày các kiến thức, kỹ năng cần thiết khi phát triển một ứng dụng iOS cơ bản thông qua việc **phân tích thiết kế và hoàn thiện từng bước** từ thiết kế giao diện màn hình, thêm vào các control phức tạp hơn (tự viết), thêm vào nhiều màn hình khác nhau, cách di chuyển giữa các màn hình, cách xử lý các sự kiện, cách ủy quyền và bổ sung các chức năng cho các đối tượng ủy quyền, phân tích thiết kế cho cơ sở dữ liệu và cuối cùng là vận dụng bản đồ trực tuyến trong ứng dụng. Do vậy sinh viên sẽ dễ dàng thực hiện và hiểu hơn các kiến thức, kỹ năng được truyền đạt.

2.5.1 Luyện tập thiết kế giao diện cơ bản và autolayout trong iOS

Trước tiên, hãy tạo một Project mới có tên FoodManagement2020 (include Test), với một màn hình đơn! Trong đó hãy luyện tập thiết kế giao diện màn hình mới với các ràng buộc như sau: Đối tượng Text Field (placeholder text: Nhập vào “Enter meal name!”, căn giữa, Return key = Done, tùy chọn “Auto-enable Return key” được chọn để đảm bảo phím Done trên bàn phím chỉ sáng lên khi Text field có dữ liệu, ràng buộc trái 47,



ràng buộc phải 47), Image View (Rộng = Dài = 320, ràng buộc Aspect Ratio để đảm bảo tỷ lệ ảnh không bị thay đổi, Intrinsic Size chọn Placeholder với chiều dài 320, chiều rộng 320 để đảm bảo ảnh co dãn theo kích thước thực khi chạy), Button (Title = Go to Map, ràng buộc lề trái 100, lề phải 100) để có kết quả như hình 2.5.1.1.

Lưu ý: Khi đưa các đối tượng vào Stack View trong chế độ Autolayout để Spacing của Stack View là 8, Allignment là Center, Distribution là Fill.

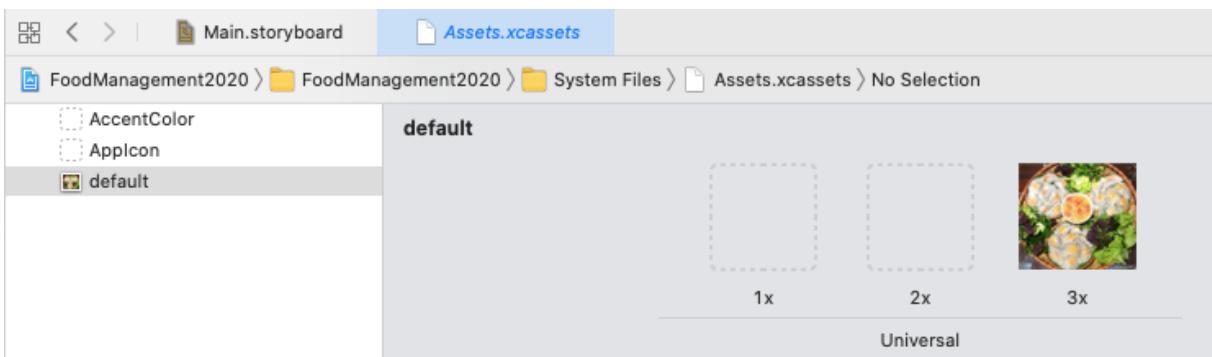
Sau khi đưa các đối tượng vào Stack view xong mới thiết lập các ràng buộc cho các đối tượng đó như lề trái, lề phải, Aspect Ratio, lựa chọn Placeholder cho Intrinsic Size của Image.

Khi chạy thử chương trình, Image view sẽ không xuất hiện do chưa có ảnh. Để thử đưa ảnh Default.

Hình 2.5.1.1 Giao diện màn hình chi tiết của một món ăn

Hướng dẫn đưa ảnh Default vào ứng dụng iOS:

Chọn Assets.xcassets trong Navigation Area => Add a group or image set => Đổi tên images set là **default** => Kéo thả ảnh tương ứng vào ô 3x (Hình 2.5.1.2).



Hình 2.5.1.2 Đưa bộ ảnh default vào ứng dụng iOS

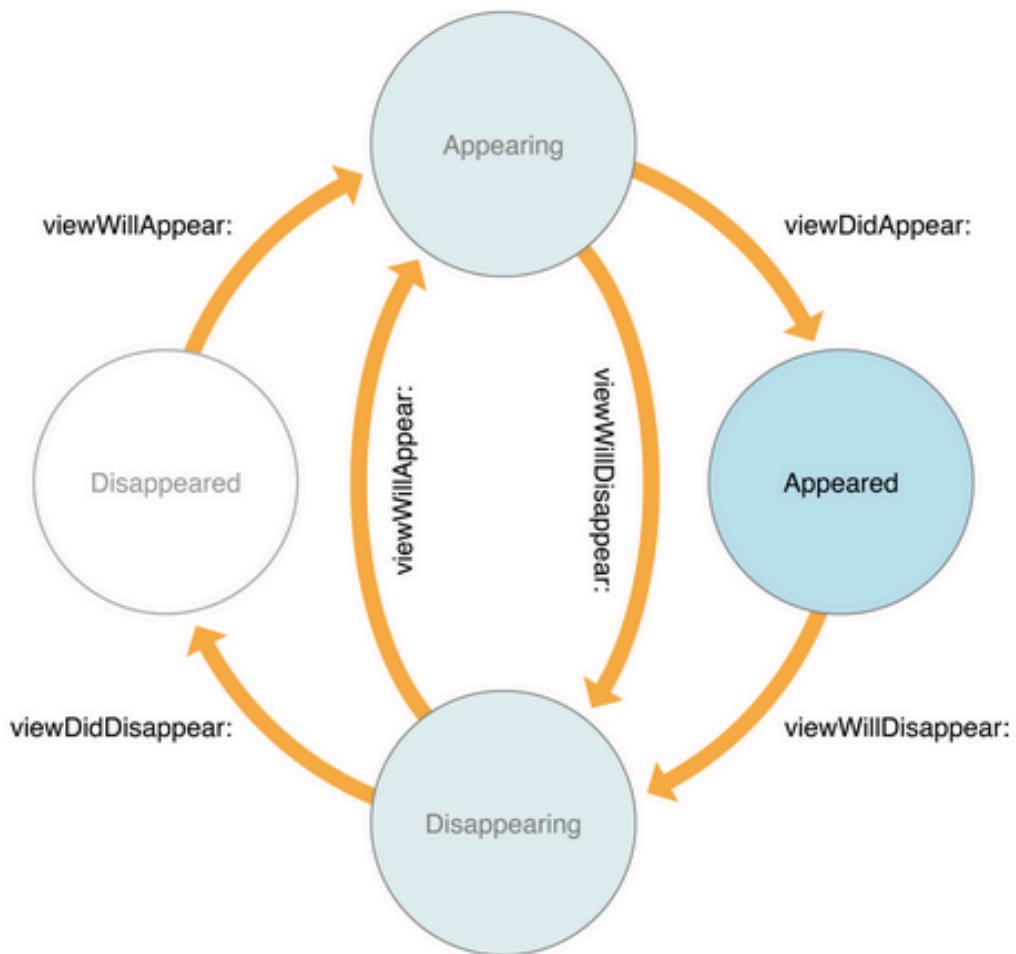
Lưu ý: Trong iOS, chỉ có 3 dạng màn hình với 3 chế độ phân giải khác nhau. Nếu muốn hỗ trợ trên cả 3 độ phân giải màn hình, cần đưa đủ cả bộ gồm 3 ảnh vào các ô 1x, 2x và 3x. Trong ví dụ này chúng ta đang chạy trên máy ảo độ phân giải 3x.

Để đưa bộ ảnh mới tạo vào image view => Chọn image view trong storyboard => Vào Attributes Inspector => Trong mục image chọn bộ ảnh **default**.

Bài tập: Chạy lại chương trình và cho nhận xét! Hãy thực hiện bài tập số 5 của chương 2 sao cho đối tượng text field và đối tượng image view đều có cùng lề trái và phải là 8.

2.5.2 Xử lý sự kiện với các Component cơ bản

Trước khi tiến hành viết code xử lý sự kiện cho các đối tượng, ta cần tìm hiểu về vòng đời của một View controller trong ứng dụng iOS. Cũng giống như Activity trong Android, một Controller trong iOS cũng có vòng đời riêng và nó tự động gọi một số hàm callbacks tại những thời điểm nhất định mỗi khi Controller đó thay đổi trạng thái (Hình 2.5.2.1 mô tả các trạng thái cơ bản của một Controller).



Hình 2.5.2.1 Các trạng thái và cách chuyển trạng thái của một Controller

Vòng đời của một Controller trong iOS:

- **Phương thức viewDidLoad()**: Phương thức này được gọi khi giao diện của Controller được khởi tạo và lấy vào từ storyboard. Phương thức cũng thường được sử dụng để thiết lập thêm các yêu cầu cấu hình bổ sung cho giao diện của Controller. Phương thức này chỉ được gọi duy nhất một lần lúc khởi tạo Controller.
- **Phương thức viewDidAppear()**: Được gọi ngay **trước** khi giao diện của Controller được đưa vào app's view hierarchy. Như vậy, phương thức này thường được dùng để thực hiện các công việc trước khi giao diện của Controller được hiển thị lên màn hình điện thoại iPhone.
- **Phương thức viewDidDisappear()**: Được gọi ngay **sau** khi giao diện của Controller được đưa vào app's view hierarchy. Như vậy, phương thức này thường được dùng để thực hiện các công việc **ngay sau** khi giao diện của Controller được hiển thị lên màn hình điện thoại iPhone (như lấy dữ liệu, thực hiện hoạt cảnh...).
- **Phương thức viewWillDisappear()**: Được gọi ngay **trước** khi giao diện của Controller được xoá khỏi app's view hierarchy. Như vậy, phương thức này thường được dùng để thực hiện các công việc dọn dẹp trước khi giao diện của Controller biến mất khỏi màn hình điện thoại iPhone.
- **Phương thức viewDidDisappear()**: Được gọi ngay **sau** khi giao diện của Controller được xoá khỏi app's view hierarchy. Như vậy, phương thức này thường được dùng để thực hiện các công việc còn lại (nếu có) sau khi giao diện của Controller biến mất khỏi màn hình điện thoại iPhone.

Xử lý sự kiện cho đối tượng Text Field:

Đối tượng này cho phép chúng ta nhập tên của món ăn. Mỗi khi tap vào đối tượng này, một Software Keyboard sẽ hiện ra trên màn hình iPhone cho phép nhập tên món ăn. Bàn phím này đã được cấu hình với phím “Done” và chỉ khi trong Text field có dữ liệu thì phím “Done” mới hoạt động (enable). Hoạt động cụ thể của phím “Done” này tùy thuộc vào nhu cầu phức tạp của người sử dụng, do vậy Swift không thể viết trước chức năng xử lý cho nó, mà sẽ thực hiện với cơ chế Uỷ quyền như đã đề cập trong chương 1.

Hãy tạo một biến liên kết code dạng tham chiếu có tên **txtFoodName** với đối tượng Text field này! Để có thể thực hiện cơ chế ủy quyền của đối tượng TextField với lớp ViewController, hãy thực hiện các bước sau:

Bước 1: Cho lớp ViewController thực hiện Protocol **UITextFieldDelegate**

```
class ViewController: UIViewController, UITextFieldDelegate {...
```

Bước 2: Thực hiện việc ủy quyền trong hàm **viewDidLoad()**:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Delegation of the TextField
    txtFoodName.delegate = self
}
```

Bước 3: Tiến hành viết các hàm đã được ủy quyền xử lý: Với Textfield có 8 phương thức được ủy quyền. Với ứng dụng này chúng ta sẽ định nghĩa cho hai phương thức sau:

```
textFieldShouldReturn(_ textField: UITextField) -> Bool
textFieldDidEndEditing(_ textField: UITextField)
```

Trước tiên cần hiểu rõ hoạt động của đối tượng TextField: Mỗi khi người sử dụng tap trên đối tượng TextField, nó sẽ tự động trở thành “**The first responder**” (Nghĩa là đối tượng được ưu tiên đầu tiên trong việc tiếp nhận các sự kiện như **key events**, **motion events**, **action messages**, ...) và khi đó iOS sẽ hiển thị bàn phím để bắt đầu soạn thảo. Mỗi khi người dùng muốn kết thúc việc soạn thảo (gõ return hoặc tap vào phím “Done” trên bàn phím máy iPhone => Phương thức **textFieldShouldReturn** sẽ được gọi) thì Textfield cần thoát khỏi trạng thái ưu tiên đó:

```
//MARK: Textfield Delegate Functions
func textFieldShouldReturn(_ textField: UITextField) -> Bool {
    // Hide the keyboard
    txtFoodName.resignFirstResponder()
    return true // Luôn xử lý mỗi khi người dùng tap “Done”
}
```

Ngoài ra, mỗi khi người dùng kết thúc việc soạn thảo, thì dữ liệu đã nhập sẽ được hiển thị ra màn hình Console (Tạm thời). Phương thức **textFieldDidEndEditing** sẽ được gọi ngay sau khi phương thức **textFieldShouldReturn** được gọi:

```
func textFieldDidEndEditing(_ textField: UITextField) {
    print("Name of the Food is \(textField.text!)")
```

Bài tập: Hãy thực hiện chương trình và cho nhận xét!

Xử lý sự kiện cho đối tượng ImageView:

Trong iOS, các đối tượng giao diện View được chia thành hai loại là các **Views** và các **Controls**. Điểm khác biệt giữa chúng là khả năng tương tác với người sử dụng: Các đối tượng là View khi liên kết code chúng ta chỉ có thể liên kết dưới dạng tham chiếu (không có liên kết dưới dạng hành vi – Action) và như vậy không thể xử lý trực tiếp các thao tác tương tác (như tap, kéo thả, ...) của người sử dụng. Ngược lại, các đối tượng Controls cho phép chúng ta có thể liên kết code cả hai dạng tham chiếu và hành vi (như đã làm với các Button, textfield...). Có thể hiểu Control như một trường hợp đặc biệt của view.

Trong ứng dụng này, chúng ta muốn mỗi khi người dùng Tap vào vùng của ImageView sẽ cho phép gọi đến hàm thực hiện việc tìm và lựa chọn một ảnh bất kỳ từ thư mục ảnh của iPhone. Tuy nhiên, ImageView lại là một loại View chứ không phải một Control, nên ta chỉ có thể tạo các liên kết code dạng tham chiếu, còn các liên kết code kiểu hành vi thì cần phải thực hiện gián tiếp thông qua một đối tượng **GestureRecognizer**. Đối tượng **GestureRecognizer** này có thể được “gắn” với một view bất kỳ và sẽ nhận trực tiếp các sự kiện tương tác từ người dùng thay cho đối tượng View, nhận dạng các hành vi đó (Tap, swipe, pinch, rotation...) và thực hiện các hành vi tương ứng (Nghĩa là nó đã biến một view giống như một Control).

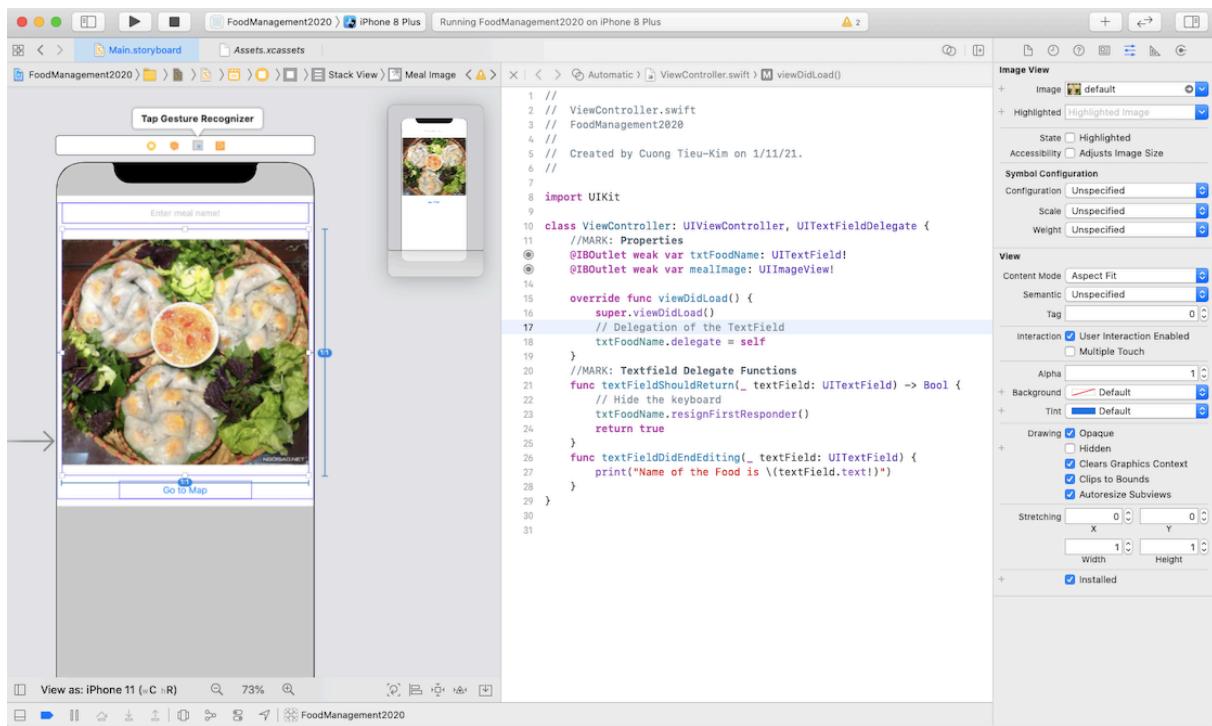
Bước 1: Trước tiên, chúng ta tạo liên kết code dạng tham chiếu với đối tượng imageview (với tên mealImage) để có thể thay đổi image của nó thành ảnh mới lấy từ thư viện ảnh:

```
@IBOutlet weak var mealImage: UIImageView!
```

Bước 2: Tạo liên kết code hành vi gián tiếp thông qua Gesture Recognizer.

Trước tiên lựa chọn đối tượng ImageView, tìm đến Attributes Inspector và đánh chọn vào lựa chọn User Interaction Enabled. Điều này sẽ cho phép đối tượng ImageView có thể nhận các tương tác từ người dùng.

Tiếp theo, thêm đối tượng Gesture Recognizer cho ImageView dựa trên các thao tác sau: Tìm trong thư viện đối tượng UITapGestureRecognizer (Có thể nhận dạng hành vi Tap của người dùng) => Kéo và thả lên trên ImageView. Hãy để ý trên thanh điều khiển của máy ảo xuất hiện thêm đối tượng Tap Gesture Recognizer (Hình 2.5.2.2).



Hình 2.5.2.2 Kết nối đối tượng Tap Gesture Recognizer với ImageView

Cuối cùng, liên kết code dạng hành vi với đối tượng Tap Gesture Recognizer này với phương thức có tên **pickUpImage** và trong hàm thêm dòng lệnh để Test thử:

```
//MARK: Tap Gesture Recognizer Action
@IBAction func pickUpImage(_ sender: UITapGestureRecognizer) {
    print("The Image view is tapped!")
}
```

Chạy chương trình và thử Tap lên đối tượng Image view, quan sát kết quả.

Xử lý sự kiện cho đối tượng Button “Go to Map”:

Chức năng này sẽ được thực hiện sau. Tuy nhiên, ta có thể liên kết code trước:

```
//MARK: Go to Map Action
@IBAction func goToMap(_ sender: UIButton) {
    print("This function is implemented later!")
}
```

2.5.3 Làm việc với UITapGestureRecognizer

Trong phần này chúng ta sẽ tiếp tục xử lý việc lấy ảnh từ thư viện ảnh của iPhone mỗi khi người dùng Tap lên đối tượng ImageView. Với các ứng dụng iOS điều này có thể dễ dàng thực hiện thông qua đối tượng **UIImagePickerController**. Cũng giống như TextField, đối tượng Image Picker cũng cho phép nơi sử dụng tự định nghĩa các hành vi đối với ảnh dựa trên **UIImagePickerControllerDelegate**. Ngoài ra, vì đối tượng

Image Picker sẽ được hiển thị mỗi khi tìm và lấy ảnh từ thư viện, do đó ứng dụng cũng cần các chức năng liên quan đến Navigation và cần thực hiện Protocol **UINavigationControllerDelegate**.

Bước 1: Thực hiện các Protocol cần thiết

```
class ViewController: UIViewController, UITextFieldDelegate,  
UIImagePickerControllerDelegate, UINavigationControllerDelegate {...}
```

Bước 2: Thực hiện chức năng chọn và lấy ảnh

Che dòng lệnh Test trong hàm **pickUpImage** đã tạo trước đó và thêm vào các dòng lệnh sau để thực hiện việc lấy ảnh mỗi khi Tap lên Image view:

```
//MARK: Tap Gesture Recognizer Action  
@IBAction func pickUpImage(_ sender: UITapGestureRecognizer) {  
    //print("The Image view is tapped!")  
  
    // 1. Hide the Keyboard of Textfield  
    txtFoodName.resignFirstResponder()  
    // 2. Declare object of Image Picker Controller  
    let imagePicker = UIImagePickerController()  
    // 3. Config the images source for the picker  
    imagePicker.sourceType = .photoLibrary  
    // 4. Delegation of Image picker's Functions  
    imagePicker.delegate = self  
    // 5. Present the image picker controller to screen  
    present(imagePicker, animated: true, completion: nil)  
}
```

Lưu ý: Nếu sử dụng Xcode phiên bản trước 9.0 thì để truy xuất vào thư viện ảnh của iPhone cần cấp quyền truy cập trong file Info.plist: Mở file Info.plist => Add new item vào cuối file => Chọn “Privacy - Photo Library Usage Description” trong danh sách sổ xuống => String => Thêm vào mô tả cho quyền (Ví dụ như: Allow to access the Photo Library of the iPhone!).

Bài tập: Chạy chương trình và cho nhận xét!

Bước 3: Viết các hàm xử lý việc lấy ảnh cho đối tượng Image Picker

Đó là các hàm được ủy quyền với đối tượng Image Picker và chúng ta cần tự viết thao tác cho các hàm đó. Ở đây, hàm **imagePickerControllerDidCancel** luôn hoạt động ở chế độ mặc định nên không cần viết lại (ngoại trừ chúng ta muốn thay đổi hành vi của nó). Ta chỉ cần viết lại hàm **imagePickerController** để xử lý việc lấy ảnh vào ứng dụng.

```

//MARK: Image Picker Controller Delegation's Functions
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {
    guard let selectedImage = info[.originalImage] as? UIImage else {
        print("Can not read the image!")
        return
    }
    // Set the selected image to the image view
    mealImage.image = selectedImage
    // Hide the image picker controller
    dismiss(animated: true, completion: nil)
}

```

Mỗi khi ảnh được đổi tượng Picker lấy về sẽ chưa trong Dictionary info, dựa vào đó chúng ta lấy ảnh gốc về, ép kiểu về dạng UIImage và đưa nó vào trong imageview.

2.5.4 Xây dựng Control mới cho ứng dụng

Với ứng dụng quản lý món ăn, chúng ta cần một đối tượng control để rating cho các món ăn khác nhau có dạng như sau:



Trong phát triển ứng dụng trên iOS, có nhiều cách khác nhau để tạo một control mới như vậy, một trong những cách đó là xây dựng một đối tượng mới dựa trên những đối tượng đã có. Một trong những thiết kế dễ nhất trong trường hợp này là xây dựng đối tượng kế thừa từ lớp StackView (theo chiều ngang), trong đó mỗi ngôi sao có thể là một image view hoặc một button. Ở đây ta sẽ sử dụng các Button.

Bước 1: Xây dựng lớp đối tượng RatingControl

Tạo file mới (Cocoa Touch class) có tên **RatingControl** kế thừa từ lớp **UIStackView** (Hình 2.5.4.1) và nội dung lớp sau khi tạo có dạng:

```

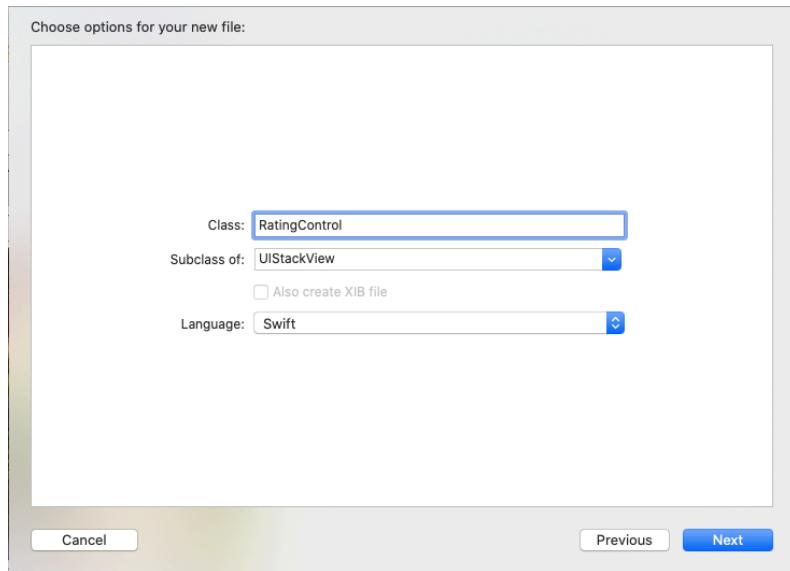
import UIKit

class RatingControl: UIStackView {
    // Code to initiate the control here!
}

```

Khi một View được xây dựng, sẽ có hai cách để tạo đối tượng view đó: Tạo bằng câu lệnh trong code hoặc tạo bằng load đối tượng từ storyboard. Do vậy, mỗi đối tượng view mới khi thiết kế cũng cần có hai phương thức khởi tạo tương ứng **init(frame:)** cho việc tạo đối tượng dùng lệnh và **init?(coder:)** cho việc load view từ storyboard.

Với Control chúng ta đang xây dựng thì cả hai phương thức này đều sẽ cùng gọi đến một phương thức chung **setUpButtons** cho việc xây dựng nên Control đó.



Hình 2.5.4.1 Tạo lớp RatingControl mới

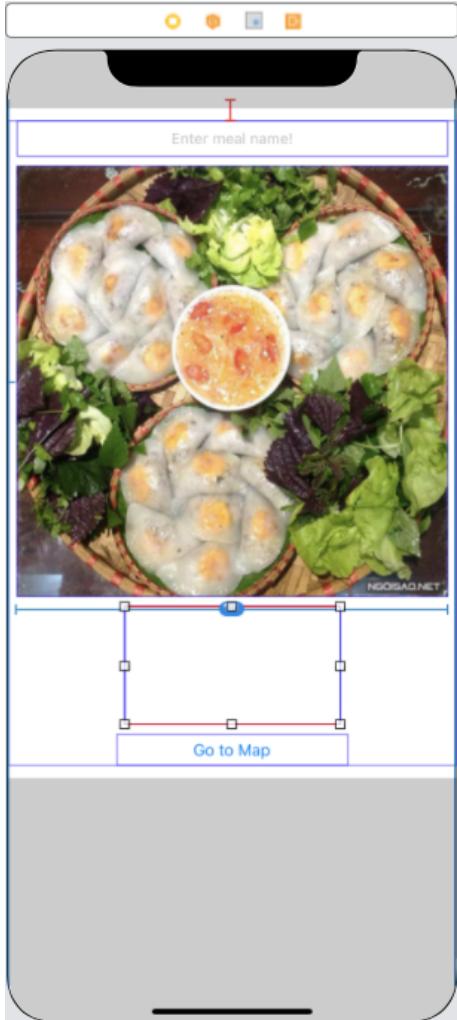
Kết quả bước 1 sẽ có dạng như sau:

```
class RatingControl: UIStackView {
    //MARK: Initialization
    override init(frame: CGRect) {
        super.init(frame: frame)
        setUpButtons()
    }
    required init(coder: NSCoder) {
        super.init(coder: coder)
        setUpButtons()
    }
    //MARK: Create properties of the Control
    func setUpButtons() {
    }
}
```

Bước 2: Thiết lập giao diện cho Control mới

Kéo thả một đối tượng StackView theo phương ngang vào ngay trước đối tượng button “Go to Map” và ngay sau đối tượng ImageView với cấu hình như sau: Vào Identity Inspector => Trong mục Class chọn RatingControl; Trong Attributes Inspector chọn Spacing = 8. Kết quả như Hình 2.5.4.2. Như vậy đối tượng StackView trong storyboard đã được kết nối với lớp RatingControl và sẽ được điều khiển bởi lớp này.

Bước 3: Từng bước thêm các button và sự kiện cho RatingControl



Trước tiên, để cho dễ hiểu chúng ta tạm thêm một button vào trong RatingControl bằng cách thêm các lệnh sau vào phuong thức setUpButtons:

```
let button = UIButton()
button.backgroundColor = UIColor.red
```

Như chúng ta đã biết, khi tạo button bằng code thì nó sẽ gọi đến hàm khởi tạo **init(frame:)** của UIButton và gán cho đối tượng mới tạo kích thước 0. Do vậy cần thiết phải điều chỉnh kích thước của Button bằng cách thêm vào các ràng buộc về chiều rộng (**button.widthAnchor**) và chiều cao (**button.heightAnchor**) của đối tượng và cuối cùng là đưa đối tượng mới tạo vào trong StackView. Có hai phương thức là **addSubview** và **addArrangedSubview**. Vì thứ tự của các button của RatingControl có vai trò khác nhau nên ta dùng phương thức thứ 2: **addArrangedSubview**.

Hình 2.5.4.2 Layout của RatingControl

Phương thức setUpButtons sẽ có dạng như sau:

```
//MARK: Create properties of the Control
func setUpButtons() {
    let button = UIButton()
    button.backgroundColor = UIColor.red
    // Setup constraints to the layout
    button.heightAnchor.constraint(equalToConstant: 44.0).isActive = true
    button.widthAnchor.constraint(equalToConstant: 44.0).isActive = true
    // Add the button to the StackView
    addArrangedSubview(button)
}
```

Chạy chương trình để xem kết quả và cho nhận xét!

Bài tập: Điều chỉnh hàm **setUpButtons** sao cho có thể thêm vào **n** button với kích thước **k** cho trước, với **n** và **k** là hai tham số.

Gợi ý: dùng **for..in** và biến **n** có kiểu Int, biến **k** có kiểu là CGSize.

Thêm sự kiện cho mỗi button: Mỗi khi một button được tap, thì nó sẽ gọi đến một hàm để thực hiện một công việc nhất định nào đó. Điều này dễ dàng thực hiện khi các button được tạo ra bởi việc kéo-thả các đối tượng view từ thư viện đối tượng vào storyboard. Tuy nhiên các đối tượng được tạo ra bằng code sẽ phức tạp hơn.

Trước tiên, xây dựng hàm sẽ được gọi mỗi khi một rating button được tap:

```
//MARK: Button's Action
private func ratingButtonTapped(button: UIButton) {
    print("rating button pressed!")
}
```

Trước khi add một button vào trong StackView thêm vào dòng lệnh sau:

```
// Add action to the rating button
button.addTarget(self, action: #selector(ratingButtonTapped(button:)),
for: .touchUpInside)
```

Đây là cách thêm các action vào đối tượng Control bằng code trong iOS với Action là một Selector trả đến phương thức sẽ được gọi khi sự kiện sau for: xuất hiện (.touchUpInside). Lưu ý: Sau khi thêm dòng lệnh trên thì sẽ có yêu cầu điều chỉnh và cần thêm chỉ báo `@objc` vào trước định nghĩa của hàm được gọi trong Selector.

Chạy thử chương trình và kiểm tra kết quả!

Bước 4: Hoàn thiện RatingControl

Trong bước này chúng ta sẽ hoàn thiện chức năng đánh giá (rating) và giao diện thật (các ngôi sao với các trạng thái khác nhau) của Control mới tạo RatingControl.

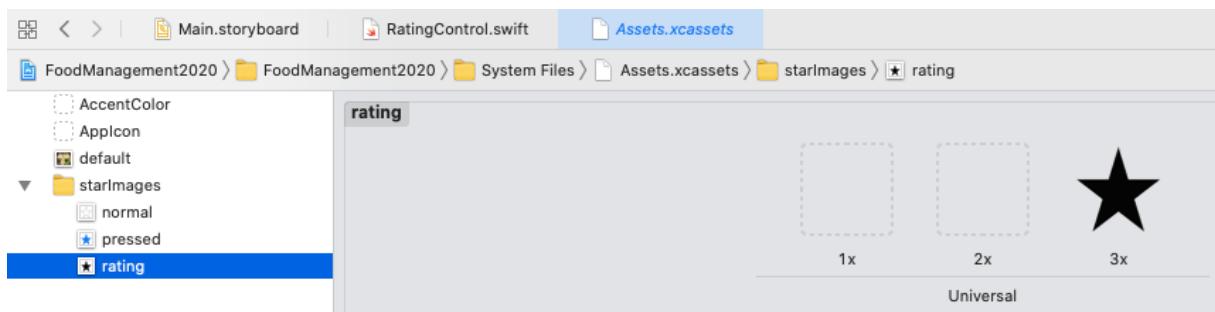
Để thực hiện chức năng rating của lớp rõ ràng chúng ta cần hai biến sau để quản lý chính các button trong StackView và giá trị **rating** hiện tại của RatingControl:

```
//MARK: Properties
private var ratingButtons = [UIButton]()
var ratingValue: Int = 0
```

Khi đó, cần thêm lệnh sau vào hàm `setUpButtons` (ngay sau lệnh dùng để thêm button vào trong StackView) để đảm bảo rằng mỗi khi tạo một button mới đồng thời ta cũng đẩy nó vào mảng để dễ dàng quản lý sau này (khi tính giá trị rating):

```
// Add the new button into array of Buttons named ratingButtons above
ratingButtons.append(button)
```

Để mỗi button có dạng các hình ngôi sao, ta cần đưa các bộ ảnh tương ứng với các trạng thái của button vào trong thư mục Assets của ứng dụng. Trong ứng dụng này, mỗi button sẽ có 3 trạng thái cơ bản: Bình thường , được nhấn và được rating . Do vậy cần đưa vào 3 bộ ảnh tương ứng với các trạng thái của button (Xem mục 2.5.1 để biết cách đưa các bộ ảnh vào trong ứng dụng iOS với lưu ý cả ba bộ ảnh đều được gom chung trong một thư mục để dễ quản lý – **starImages**). Kết quả như hình 2.5.4.3.



Hình 2.5.4.3 Ba bộ ảnh cho đối tượng RatingControl

Muốn thay đổi giao diện cho các button từ các ô vuông màu đỏ thành các ngôi sao, hãy thay câu lệnh: `button.backgroundColor = UIColor.red` trong hàm `setUpButtons` thành các câu lệnh sau:

```
// Set images to the button
button.setImage(normal, for: .normal)
button.setImage(pressed, for: .highlighted)
button.setImage(rating, for: .selected)
```

Và trước vòng lặp `for...in` thêm vào các dòng lệnh sau để load ảnh từ Assets vào các biến trước khi đưa chúng vào các button:

```
// Load images from starImages
let normal = UIImage(named: "normal")
let pressed = UIImage(named: "pressed")
let rating = UIImage(named: "rating")
```

Bài tập: Thủ chạy chương trình, tap trên các button bất kỳ và cho nhận xét!

Với trạng thái **normal** và trạng thái khi một button bị tap (có trạng thái tên **highlighted**) thì hình ảnh được thiết lập cho button hoạt động bình thường. Tuy nhiên, trạng thái khi chúng ta muốn rating cho một món ăn (tên trạng thái trong iOS là **selected**) thì chưa thấy xuất hiện. Muốn thực hiện điều đó, cần điều chỉnh trong hàm `ratingButtonTapped` bằng cách thay thế dòng lệnh Test bằng những dòng lệnh dưới đây nhằm thay đổi giá trị của biến **ratingValue** tuỳ theo button nào đã được tap.

```

//MARK: Button's Action
@objc private func ratingButtonTapped(button: UIButton) {
    // Get the index of the pressed button in our array
    if let index = ratingButtons.firstIndex(of: button) {
        // Calculate the ratingValue of the pressed button
        let selectedRating = index + 1
        // Change value of ratingValue
        if selectedRating == ratingValue {
            ratingValue -= 1
        } else {
            ratingValue = selectedRating
        }
    }
}

```

Sau khi giá trị của biến ratingValue thay đổi, chúng ta cần cập nhật lại trạng thái cho các rating buttons cho phù hợp. Vì thao tác này thực hiện nhiều nơi, nên chúng ta viết chúng dưới dạng một hàm như sau:

```

private func updateButtonStates(){
    // Browse the array with position of each button
    for (index, button) in ratingButtons.enumerated() {
        // Set all buttons having their index < ratingValue to selected state
        button.isSelected = index < ratingValue
    }
}

```

Trước tiên hàm này sẽ được gọi ở cuối phương thức setUpButtons và cuối phương thức ratingButtonTapped. Thực hiện chương trình và cho nhận xét! Điều chỉnh giá trị tham số biến ratingValue khác nhau, chạy lại chương trình và cho nhận xét!

2.5.5 Thêm thuộc tính vào Attributes Inspector

Trước tiên, chúng ta thấy rằng do các button được sinh bằng code nên trong màn hình Storyboard chúng ta không thấy được các button mà chỉ nhìn thấy chúng khi chạy chương trình. Muốn thấy trước được kết quả trong màn hình thiết kế, hãy thêm chỉ báo @IBDesignable vào trước khai báo của lớp RatingControl như sau:

```
@IBDesignable class RatingControl: UIStackView {...}.
```

Chỉ báo này cho phép Interface Builder tạo ra và vẽ một bản copy của đối tượng trực tiếp trên màn hình giao diện do vậy ta có thể nhìn thấy trước kết quả giống như khi chạy chương trình. Tuy nhiên, thay đổi này chưa hiệu ứng ngay trong trường hợp của chúng ta do ở trong chế độ @IBDesignable thì các câu lệnh load ảnh sau không hoạt động:

```
let normal = UIImage(named: "normal")
```

```

let pressed = UIImage(named: "pressed")
let rating = UIImage(named: "rating")

```

Do trong chế độ này khi tạo và vẽ bản copy là thực hiện trong **Interface Builder** nên nó không hiểu **assets catalog** ở đâu. Chú ý rằng **assets catalog** được đặt trong **main bundle** của ứng dụng. Do vậy cần gọi tường minh như sau:

```

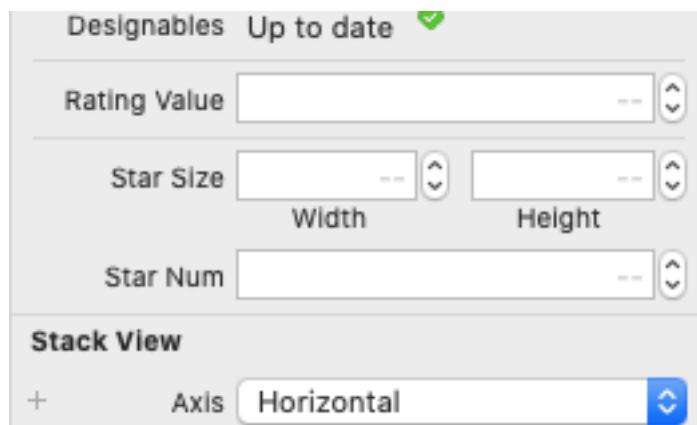
// Get the main bundle
let bundle = Bundle(for: type(of: self))
// Load images
let normal = UIImage(named: "normal", in: bundle, with: .none)
let pressed = UIImage(named: "pressed", in: bundle, with: .none)
let rating = UIImage(named: "rating", in: bundle, with: .none)

```

Trong lớp RatingControl có 3 tham số là **ratingValue**, **starSize** và **starNum** mà người dùng có thể thay đổi mỗi khi chạy ứng dụng. Các tham số này trong iOS hoàn toàn có thể đưa trực tiếp vào trong các bảng thuộc tính của đối tượng (trong **Attributes Inspector**) và người dùng có thể thay đổi giá trị các thuộc tính trực tiếp từ trong bảng này. Để có thể thực hiện được điều đó hãy làm như sau.

Bước 1: Khai báo các thuộc tính trong bảng Attributes Inspector.

Đưa vào trước khai báo của các biến đó chỉ báo sau: `@IBInspectable`. Biên dịch lại và hãy lựa chọn đối tượng RatingControl trên Storyboard sẽ thấy trong Attributes Inspector xuất hiện 3 biến mới tạo (Hình 2.5.4.4).



Hình 2.5.4.4 Thuộc tính mới trong Attributes Inspector

Tuy nhiên, người sử dụng vẫn chưa thay đổi được giá trị của mỗi tham số trực tiếp trên bảng Attributes Inspector này mà đang sử dụng giá trị mặc định cho từng loại biến (dưới – trong mỗi ô thuộc tính chỉ ra rằng nó đang sử dụng giá trị mặc định).

Muốn có thể sửa trực tiếp trên bảng thuộc tính Attributes Inspector thì mỗi khi giá trị ở đó thay đổi cần cập nhật lại thuộc tính biến tương ứng bên trong RatingControl. Trong iOS để làm được điều đó mỗi thuộc tính cần cập nhật cần được thêm vào một **Property observer**. Property observer sẽ được gọi mỗi khi giá trị của một thuộc tính thay đổi. Sửa lại khai báo của 3 tham số trong RatingControl để thêm vào Property observer có tên **didSet** như sau sao cho mỗi khi giá trị của biến thay đổi thì gọi lại hàm setUpButtons (Nghĩa là thiết lập lại các button cho RatingControl).

```
@IBInspectable var ratingValue: Int = 0 {
    didSet {
        setUpButtons()
    }
}
@IBInspectable private var starSize: CGSize = CGSize(width: 44.0,
height: 44.0) {
    didSet {
        setUpButtons()
    }
}
@IBInspectable private var starNum: Int = 5 {
    didSet {
        setUpButtons()
    }
}
```

Bài tập: Thực hiện chương trình, thay đổi các tham số trong bảng Attributes Inspector, quan sát kết quả và giải thích! Hãy đưa giải pháp điều chỉnh!

Gợi ý: Cần xoá đi các button cũ trong mảng và trong StackView mỗi khi hàm setUpButtons được gọi lại. Việc xoá button từ StackView cần qua hai bước: Trước tiên sẽ xoá button từ danh sách các views được quản lý bởi StackView (để báo cho StackView biết không cần phải tính toán kích thước và vị trí của button trong Stack nữa), tuy nhiên nó vẫn là Subview của StackView đó. Tiếp theo là xoá button hoàn toàn khỏi StackView. Và cuối cùng mới xoá các button khỏi mảng button.

```
// Clear the old button
for button in ratingButtons {
    removeArrangedSubview(button)
    button.removeFromSuperview()
}
ratingButtons.removeAll()
```

Chạy lại chương trình, điều chỉnh các giá trị thuộc tính trực tiếp trong lớp RatingControl và trên bảng thuộc tính Attributes Inspector! Cho nhận xét!

2.5.6 Table view

Ứng dụng quản lý món ăn trước tiên cần hiển thị danh sách các món ăn, mỗi món ăn có dạng như hình 2.5.6.1 (Có hình ảnh món ăn; Tên món ăn và đánh giá của món ăn đó).



Hình 2.5.6.1 Thiết kế giao diện một phần tử trong danh sách các món ăn

Mỗi khi nhấp chọn vào một món ăn cụ thể mới chuyển sang màn hình chi tiết như đã thiết kế trong các phần trước (Giống như ListView trên Android). Trong iOS, để thực hiện màn hình dạng này cần tạo một TableView được điều khiển bởi một UITableViewController.

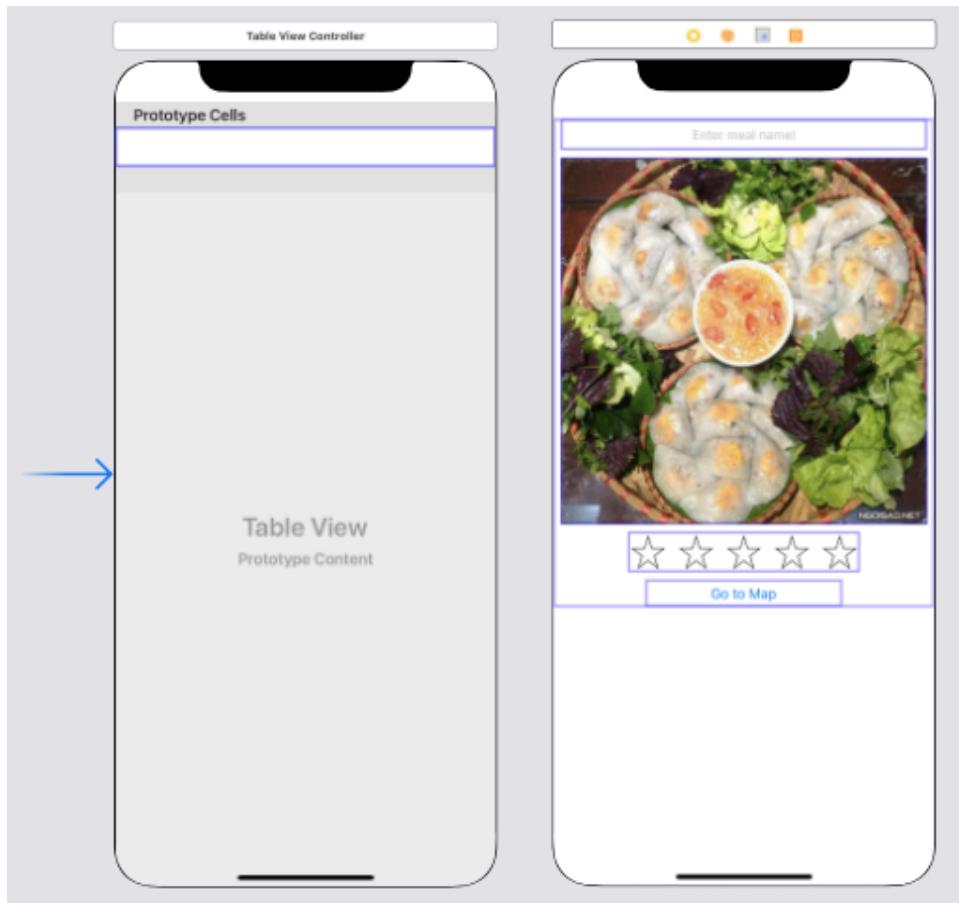
Trong iOS, mỗi màn hình (giống như một Activity trong Android) sẽ có một giao diện **Layout controller** (thường trong storyboard) và một file ***Controller.swift** tương ứng để điều khiển mọi hoạt động của màn hình đó (mỗi màn hình trong iOS được gọi là một **Scene**). Để có thể thêm một màn hình dạng Table view trong iOS (giống với ListView trong Android), ta cần thực hiện những thao tác sau:

Bước 1: Tạo giao diện màn hình và lớp điều khiển

- Tìm trong thư viện đối tượng **Table View Controller** và kéo thả vào trong storyboard.
- Kéo thả **mũi tên điểm vào** từ màn hình **View Controller** sang màn hình **Table View Controller** vừa mới tạo (Hình 2.5.6.1) để đảm bảo rằng màn hình mới tạo được chạy đầu tiên khi thực hiện chương trình.
- Tạo lớp Controller mới có tên **MealViewController.swift** kế thừa từ lớp **UITableViewController** và dùng ngôn ngữ Swift.
- Chọn màn hình Table View Controller trên storyboard, truy xuất vào bảng Identity Inspector và trong mục Class lựa chọn **MealViewController** để gắn kết giao diện mới tạo với lớp Controller mới tạo với nhau.

Bước 2: Cấu hình cho giao diện Table View Controller trên Storyboard

- Chọn màn hình mới tạo trên Storyboard => Shift + Phải chuột => Chọn Table View
- Mở bảng Size Inspector => Row Height => Nhập 90 => Enter



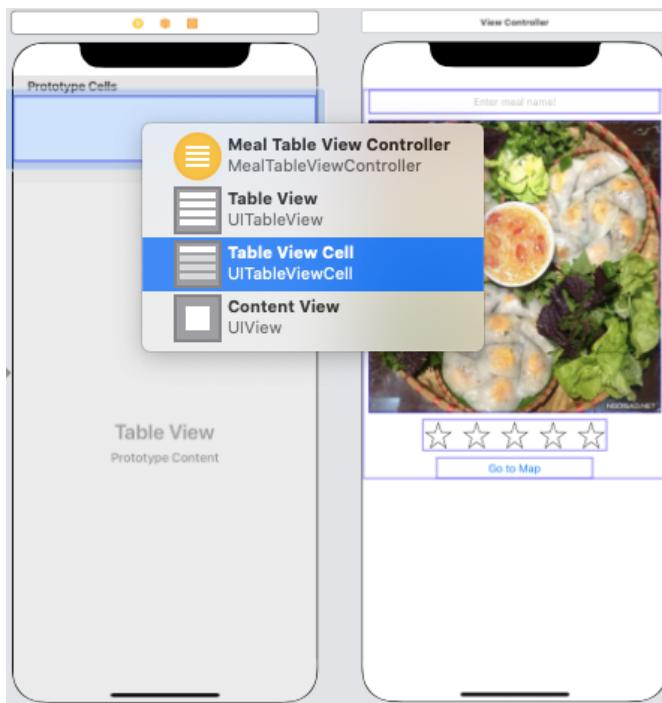
Hình 2.5.6.1 Tạo màn hình giao diện mới với Table View Controller

Bước 3: Thiết kế và cấu hình cho một phần tử của Table View (Cell)

Mỗi phần tử (Hay một hàng – Row) của một Table View trong iOS được gọi là một Cell (Trong Android được gọi là một ListView Item). Mỗi một Cell này đến lượt nó lại được điều khiển bởi lớp **UITableViewCell** (chức năng giống như hàm getView trong Adapter của ListView trong Android). Để có thể thiết kế và cấu hình cho một Cell của Table View, thực hiện những thao tác sau (giống như Customize ListView trong Android):

- Tạo lớp điều khiển cho mỗi Cell (Lớp con của lớp UITableViewCell): Tên lớp **MealTableViewCell**, kế thừa từ lớp **UITableViewCell**, ngôn ngữ Swift.
- Mở lại Storyboard, Shift + Phải chuột trên **Cell trống đơn** duy nhất phía trên màn hình giao diện của Table View Controller => Chọn Table View Cell (Hình 2.5.6.2). Cell trống đơn duy nhất đó trong iOS gọi là **Prototype Cells** (Có thể hiểu nó như một **Cell hình thức** cho lập trình viên thiết kế và cấu hình). Những gì được thiết lập cho **Prototype Cell** tại thời điểm thiết kế thì cũng sẽ được hiệu ứng sau này khi chạy chương trình cho mỗi Table View Cell (Table Rows) của Table View hiện tại. Sau khi chọn

Prototype Cell, mở Identity Inspector => Trong mục Class chọn **MealTableViewCell** để kết nối Prototype Cell này với lớp điều khiển cho mỗi Cell của Table View.



Hình 2.5.6.2 Lựa chọn Prototype Cell để thiết kế và cấu hình

Đồng thời, trong bảng Attributes Inspector, đến mục **Identifier** nhập vào **MealTableViewCell**. Đây là tên biến sẽ được dùng để tạo ra các đối tượng Cell của chính Prototype Cell sau này. Tiếp tục mở bảng Size Inspector, đến mục Row Height và nhập vào 90 => Enter.

Bước 4: Thiết kế giao diện của Prototype Cell giống với Hình 2.5.6.1

Kéo và thả đối tượng ImageView (để chứa ảnh của món ăn) từ thư viện vào trong Prototype Cell. Điều chỉnh cho ImageView nằm ở góc trái của Cell đó và thiết lập bộ ảnh default cho image view này. Tiếp theo, kéo thả một đối tượng Label (để chứa tên món ăn) từ thư viện vào Prototype Cell. Điều chỉnh kích thước cho hợp lý. Cuối cùng kéo thả một đối tượng StackView dạng ngang vào bên dưới Label và bên phải của ImageView. Lựa chọn StackView, vào Size Inspector trong mục Height nhập 44. Kéo di chuyển đối tượng cho đến khi gấp những đường nét đứt (đường căn lề) thì dừng lại. Tiếp theo trong Identity Inspector, đến mục Class chọn RatingControl. Trong Attributes Inspector, đến mục Distribution chọn Fill Equally, đến mục Spacing nhập 8. Kết quả sau thiết kế cho Prototype Cell sẽ như hình 2.5.6.3.



Hình 2.5.6.3 Thiết kế cho Prototype Cell

Bước 5: Chuẩn bị data source cho Table View

Cũng giống như trong Android, ta cần một mảng các đối tượng meal cho Table view. Trước tiên, ta cần định nghĩa cấu trúc dữ liệu để lưu trữ cho mỗi món ăn: **Meal**. Thực hiện tạo file mới kiểu Swift có tên Meal. Trong file này ta sẽ định nghĩa cấu trúc dữ liệu để chứa thông tin các món ăn như tên món ăn (mealName có kiểu là String), hình ảnh của món ăn (mealImage có kiểu là UIImage) và giá trị rating cho món ăn đó (ratingValue có kiểu là Int) với các ràng buộc: Hình ảnh của món ăn thì có thể có hoặc không, giá trị rating của món ăn có giá trị từ 0 đến 5 (0 là chưa đánh giá, 1 là 1 sao...). Tên món ăn và đánh giá phải có giá trị (không được nil). Do trong lớp món ăn này, có tên và giá trị không nil (không phải biến Optional) nên ta cần định nghĩa hàm init cho nó (Xem lại chương 1). Do khi tạo đối tượng cần có một số ràng buộc và nếu không thỏa mãn các ràng buộc đó thì không tạo được đối tượng món ăn (trả về nil). Do vậy hàm init được dùng có dạng init? (trả về nil nếu không tạo được đối tượng):

```
import UIKit
class Meal {
    //MARK: Properties
```

Do trong màn hình này người dùng không được phép điều chỉnh thông tin của món ăn (muốn điều chỉnh phải vào màn hình chi tiết món ăn đã thực hiện trước đó), nên ta cần tắt chế độ tương tác với người dùng cho đối tượng RatingControl trong Prototype Cell: Chọn đối tượng RatingControl trong Prototype Cell => Mở bảng thuộc tính Attributes Inspector => Đến mục **Interaction** => Bỏ lựa chọn “User Interaction Enable”. Chạy thử chương trình và cho nhận xét!

Với TableView trong iOS, Prototype Cell chỉ dùng để thiết kế, nó không tự động hiển thị ra màn hình. Muốn có kết quả như mong muốn, mỗi khi có một món ăn cần hiển thị ra trên màn hình TableView, nhất thiết phải tạo một đối tượng Cell từ Prototype Cell và đổ dữ liệu vào đó trước khi yêu cầu hiển thị nó ra màn hình.

```

var mealName: String
var mealImage: UIImage?
var ratingValue: Int
//MARK: Initialization
init?(name: String, image: UIImage?, rating: Int) {
    // Check the conditions
    guard !name.isEmpty else {
        return nil
    }
    guard (rating >= 0) && (rating <= 5) else {
        return nil
    }
    // Initialization of class's properties
    mealName = name
    mealImage = image
    ratingValue = rating
}

```

Bước 6: Kết nối Table View Cell với code

Mở file MealTableViewCell.swift => Kết nối code dạng tham chiếu cho 3 đối tượng trong Prototype Cell:

```

//MARK: Properties
@IBOutlet weak var mealImage: UIImageView!
@IBOutlet weak var mealName: UILabel!
@IBOutlet weak var ratingControl: RatingControl!

```

Bước 7: Hiển thị cell trên TableView

Việc hiển thị các cell trên TableView sẽ được điều khiển bởi MealTableViewController đã tạo trước đó. Trước tiên cần tạo một mảng rỗng các món ăn:

```

//MARK: Properties
var meals = [Meal]()

```

Và trong hàm viewDidLoad() tạo một món ăn mẫu để Test như sau:

```

// Create an example of meal
let image = UIImage(named: "default")
if let meal = Meal(name: "Mon Hue", image: image, rating: 3) {
    meals += [meal]
}

```

Di chuyển đến phương thức `numberOfSections` và sửa thành **return 1** (Table View có 1 Section). Trong iOS, mỗi Section trong Table View dùng để hiện thị thành nhóm các Cells khác nhau. Với ứng dụng này các cell hiện thị giống nhau (1 section). Tiếp theo, tìm đến phương thức `tableView(_:numberOfRowsInSection:)` để trả về số phần tử của

Table View cần hiển thị. Ở đây ta sẽ trả về kích thước của mảng meals. Tiếp theo di chuyển đến phương thức `tableView(_:cellForRowAt:)` và xoá bỏ cặp comment bên ngoài /* */ để sử dụng phương thức này. Điều chỉnh code trong thân phương thức:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return meals.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "MealTableViewCell"
    guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath) as? MealTableViewCell else {
        fatalError("Can not create the Cell!")
    }
    // Fetches the appropriate meal for the data source layout
    let meal = meals[indexPath.row]
    cell.mealName.text = meal.mealName
    cell.mealImage.image = meal.mealImage
    cell.ratingControl.ratingValue = meal.ratingValue
    return cell
}
```

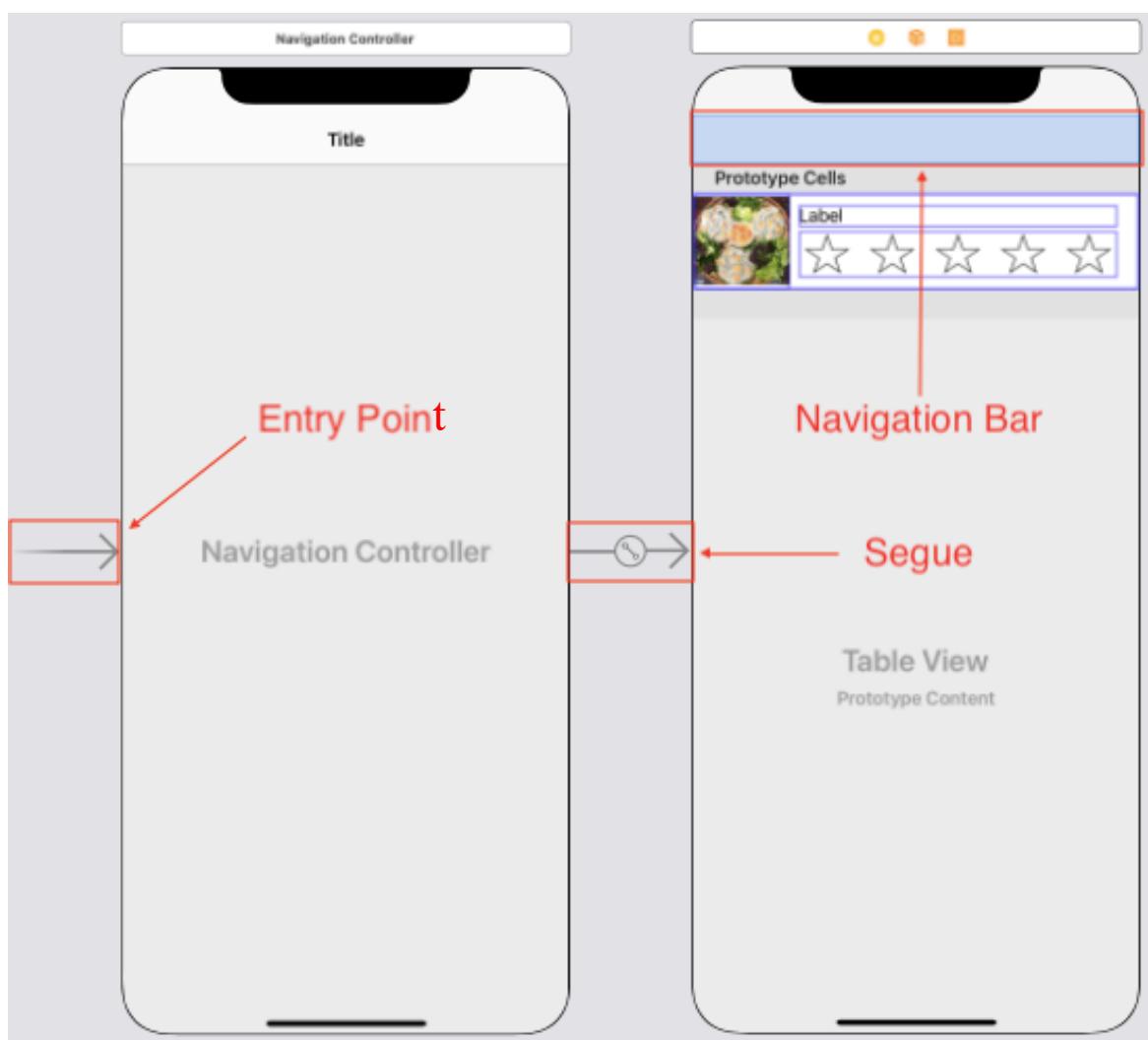
Chạy thử chương trình và cho nhận xét!

2.5.7 Navigation và truyền tham số giữa các màn hình ứng dụng

Khi một ứng dụng iOS có nhiều màn hình (nhiều Scences) khi đó ta cần thiết kế sao cho có thể dễ dàng di chuyển và truyền tham số giữa các màn hình đó với nhau. Trong iOS, việc di chuyển đó thường được quản lý bởi một đối tượng **Navigation Controller**.

Với ứng dụng Quản lý món ăn đang thực hiện, trong màn hình Table View Controller hiển thị danh sách các món ăn (ảnh, tên và đánh giá của món ăn). Chúng ta mong muốn rằng mỗi khi nhấp chọn vào một món ăn cụ thể trong danh sách món ăn hiện có, ứng dụng sẽ di chuyển tới màn hình chi tiết món ăn (đã làm trước đó) và cho phép người dùng xem hoặc điều chỉnh, cập nhật cho món ăn đó. Ngoài ra, chúng ta cũng mong muốn rằng, trong màn hình Table View Controller sẽ có một nút thêm mới món ăn, xoá món ăn trong danh sách... Việc thêm mới món ăn thực chất cũng sẽ di chuyển đến màn hình chi tiết món ăn nhưng là để cho người dùng thêm món ăn mới chứ không phải cập nhật lại thông tin cho món ăn đã có. Trước tiên, chúng ta thực hiện việc tạo mới món ăn trước với các thao tác sau:

Bước 1: Nhúng màn hình đầu tiên (Table View Controller) vào một đối tượng **Navigation Controller** bằng cách chọn Table View Controller trên storyboard => Editor => Embed In => Navigation Controller. Một đối tượng Navigation Controller được thêm vào storyboard (ngay trước màn hình Table View Controller) và mũi tên thiết lập điểm vào của ứng dụng được chuyển sang Navigation Controller, còn Table View Controller trở thành **Root View Controller** trong Navigation Stack của Navigation Controller đó (Root View Controller sẽ không bao giờ bị lấy ra khỏi Stack). Giữa Navigation Controller và Table View Controller xuất hiện một mối liên kết gọi là một **Segue** đồng thời phía trên màn hình đó xuất hiện một thanh trống gọi là **Navigation Bar** (Hình 2.5.7.1).



Hình 2.5.7.1 Sau khi nhúng Table View Controller vào Navigation Controller **Segue** là đối tượng rất quan trọng được dùng trong quá trình chuyển đổi qua lại giữa các màn hình. Còn thanh Navigation Bar dùng để chứa các điều khiển cho việc di chuyển qua lại giữa các màn hình (backward và forward). Mọi đối tượng View Controller ở

trong Navigation Stack đều có một thanh Navigation Bar của riêng nó. Trong ứng dụng này, chúng ta cần thêm vào Navigation Bar của Table View Controller một điều khiển dùng để di chuyển từ màn hình Table View Controller sang màn hình chi tiết món ăn mỗi khi chúng ta cần thêm một món ăn mới vào danh sách.

Bước 2: Thiết kế và cấu hình cho thanh Navigation Bar

Thông thường trên một thanh Navigation Bar có 3 đối tượng thường gặp: Tiêu đề (nằm ở giữa thanh Navigation Bar), nút điều khiển trái (Left Bar Item Button) và nút điều khiển phải (Right Bar Item Button). Với màn hình Table View Controller chúng ta muốn nó có tiêu đề là “Meal List”, nút điều khiển trái có tên Edit (edit cho danh sách món ăn – xoá phần tử trong danh sách) và nút điều khiển phải có tên “+” để thêm món ăn mới. Với nút điều khiển trái, do Table View Controller có sẵn các phương thức cho việc uỷ quyền, bao gồm cả việc điều chỉnh và cập nhật trong Table View. Cho nên chúng ta sử dụng đối tượng điều khiển có sẵn của Table View Controller. Trong hàm viewDidLoad() hãy thêm dòng lệnh sau:

```
// Add Left Bar Item Button on the Bar
navigationItem.leftBarButtonItem = editButtonItem
```

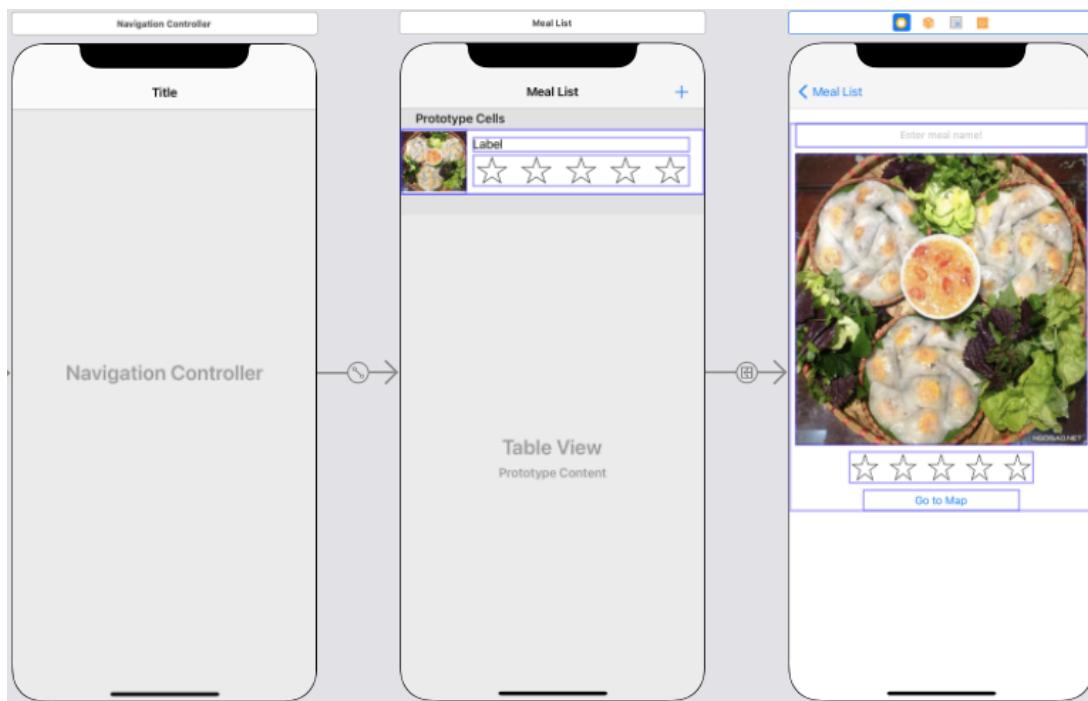
Trong iOS, mỗi View Controller sẽ có một biến navigationItem dùng để cấu hình cho Navigation Bar nếu cần thiết. Khi chạy chương trình sẽ thấy trên thanh Navigation Bar bên trái có một button **Edit** (editButtonItem là đối tượng item bar button đã được cấp sẵn cho mỗi Table View). Để có thể xoá được các phần tử (Row) trong Table View, hãy di chuyển đến hàm uỷ quyền có tên tableView(_:commit:forRowAt:) và xoá bỏ khối comment bên ngoài hàm. Trong khối lệnh sau điều kiện của if (editingStyle == .delete), ngay sau dòng comment “// Delete the row from the data source” hãy thêm lệnh: `meals.remove(at: indexPath.row)` để xoá phần tử tương ứng trong datasource của Table View.

Tiếp theo, chúng ta cần điều chỉnh cho Title của Navigation Bar bằng cách nhấp đúp lên vùng trống chính giữa của Navigation Bar, hoặc chọn Navigation Bar sau đó chọn bảng Attributes Inspector, đến mục Title và nhập vào “Meal List”. Kéo thả một đối tượng **Bar Button Item** từ thư viện đối tượng vào góc phải của thanh Navigation Bar. Chọn button đó, mở bảng Attributes Inspector, đến mục System Item chọn Add, khi đó nút button đó sẽ có biểu tượng dấu “+”.

Bước 3: Điều hướng sang màn hình chi tiết món ăn

Trước tiên, chúng ta nên đổi tên màn hình ViewController trước đó (tên mặc định khi tạo mới ứng dụng) thành màn hình có tên MealDetailController (thực hiện hai thao tác đổi tên và kết nối màn hình trên giao diện với code).

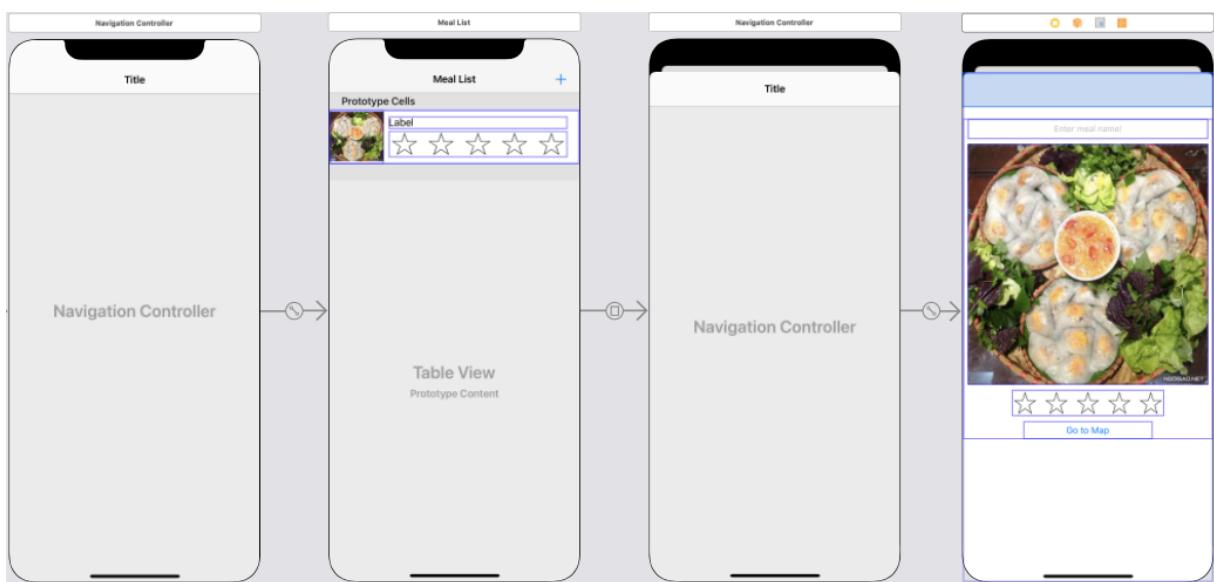
Ctrl + Drag từ nút “+” trên thanh Navigation Bar sang màn hình MealDetailController => Một bảng lựa chọn xuất hiện => Chọn Show. Khi đó giữa màn hình Table View Controller và màn hình MealDetailController xuất hiện một Segue mới (hình dáng khác với Segue giữa Navigation Controller với Table View Controller). Đồng thời, màn hình MealDetailController cũng xuất hiện một thanh Navigation Bar và có một nút điều khiển trái mang tên “< Meal List” (Dùng để quay lại màn hình có Title trên thanh Navigation Bar là “Meal List”). Chạy thử chương trình và cho nhận xét!



Hình 2.5.7.2 Màn hình MealDetail sau khi được đưa vào Navigation Stack

Khi chúng ta chọn Segue dạng “Show” thì khi đó màn hình đích (mà hình mà khi tap vào nút “+” sẽ được di chuyển đến – chính là MealDetailController) sẽ được đẩy vào trong Navigation Stack (và nằm trên đỉnh Stack). Navigation Controller sẽ luôn hiển thị màn hình đang nằm trên đỉnh Navigation Stack (và do đó MealDetailController được hiển thị ra màn hình). Khi tap vào nút “< Meal List” thì màn hình MealDetailController sẽ bị lấy ra khỏi Navigation Stack (Pop) và màn hình Meal List sẽ lại được hiển thị. Tuy nhiên, với cách hoạt động này, chưa hoàn toàn phù hợp với chức năng ta đang thiết kế:

Mỗi khi tap vào nút “+” sẽ di chuyển đến màn hình MealDetailController để cho phép người dùng tạo ra một món ăn mới (Tên, ảnh, rating) và sau khi hoàn thành sẽ gửi trả lại cho màn hình Meal List để thêm kết quả món ăn mới vào danh sách. Do vậy, kiểu di chuyển của Segue dạng Show là không phù hợp trong trường hợp này. Nhấp chọn vào biểu tượng Show Segue  trong liên kết giữa hai màn hình, mở bảng Attributes Inspector, di chuyển đến mục Kind chọn **Present Modally**, đồng thời cũng đặt tên cho Segue này là **addNewMealSegue** bằng cách di chuyển đến mục **Identifier** và nhập tên này vào đó. Với kiểu Segue dạng này thì màn hình MealDetailController sẽ không được đẩy vào trong Navigation Stack và màn hình MealDetailController cũng không được thêm vào thanh Navigation Bar, do đó việc di chuyển qua lại giữa màn hình đó với các màn hình khác sẽ do chúng ta tự thiết kế. Để có thể thêm vào thanh Navigation Bar cho màn hình MealDetailController và màn hình MapController sau này (Khi tap vào nút “Go to Map” sẽ chuyển đến màn hình hiện bản đồ trực tuyến trong ứng dụng), chúng ta cần nhúng màn hình MealDetailController vào một Navigation Controller riêng của nó. Chọn màn hình MealDetailController => Editor => Embed In => Navigation Controller. Lúc này, một màn hình Navigation Controller được thêm vào giữa màn hình Meal List và màn hình MealDetailController (MealDetailController trở thành Root trong Navigation Stack của Navigation Controller mới) và trong màn hình MealDetail cũng xuất hiện một thanh Navigation Bar mới (Hình 2.5.7.3).



Hình 2.5.7.3 Kết quả sau khi nhúng MealDetail vào Navigation Controller mới
Chạy thử chương trình và cho nhận xét!

Mới đầu khi chuyển màn hình, chúng ta muốn nó có Title là “New Meal”. Sau đó, trong màn hình MealDetailController mỗi khi người dùng nhập tên mới cho món ăn, thì nó không log ra màn hình console nữa mà tên đó sẽ được đưa trực tiếp vào trong Title của Navigation Bar. Trước tiên, nhập Title cho Navigation Bar mới với tên “New Meal”. Tiếp theo, di chuyển đến hàm `textFieldDidEndEditing` và thay thế dòng lệnh `print("Name of the Food is \(textField.text!)")` bằng câu lệnh:

```
navigationItem.title = textField.text
```

Bước 4: Thiết kế và cấu hình Navigation Bar của màn hình chi tiết món ăn

Ta muốn rằng, trong màn hình chi tiết món ăn có button trái là Cancel để quay lại màn hình trước đó mà không cần thay đổi thông tin gì trong danh sách món ăn và button phải là Save để quay lại màn hình trước đó (Meal List) và thêm vào danh sách món ăn mới. Hãy kéo thả đối tượng Bar Button Item từ thư viện vào góc trái của Navigation Bar rồi đổi System Item thành Cancel, tương tự với button bên phải Navigation Bar là Save. Với nút Cancel, liên kết code nút này dưới dạng hành vi và gõ vào dòng lệnh sau để quay lại màn hình trước đó: `dismiss(animated: true, completion: nil)`.

Bước 5: Thực hiện di chuyển và Truyền tham số giữa các màn hình

Truyền tham số giữa các màn hình là một trong những yêu cầu quan trọng khi di chuyển giữa các màn hình với nhau. Trong iOS, việc truyền tham số này có thể thực hiện khá dễ dàng nhờ vào các đối tượng Segue. Trong ví dụ của chúng ta, chúng ta mong muốn rằng, mỗi khi ở màn hình MealDetail người sử dụng tap vào nút Save thì món ăn mới do người sử dụng nhập vào sẽ được truyền sang màn hình MealList và được màn hình MealList cập nhật vào danh sách các món ăn đã có. Để làm được điều đó, trước tiên ta cần một biến thành phần meal (đối tượng của lớp datamodel **Meal**). Biến thành phần này có thể là của lớp nguồn hoặc lớp đích đều được. Trong trường hợp của chúng ta, món ăn mới thuộc lớp MealDetailController, do đó ta sẽ khai báo biến thành phần **meal** là của lớp nguồn – Lớp MealDetailController (Nó là một biến Optional). Biến này sẽ được dùng để lưu đối tượng món ăn mới và sẽ được truy xuất từ màn hình MealList sau này:

```
//MARK: Properties  
var meal: Meal?
```

Trong iOS, khi từ màn hình A (màn hình nguồn) di chuyển đến màn hình B (màn hình đích), thì trước khi việc di chuyển thực sự diễn ra, bên màn hình A luôn gọi đến một hàm có tên **prepare(for:sender:)** để cho phép lập trình viên thực hiện các thao tác chuẩn bị, lưu dữ liệu và thực hiện những công việc cần thiết khác. Di chuyển đến phần đánh dấu //MARK: Navigation Actions, bên dưới hàm cancelNewMeal gõ vào prepare rồi chọn hàm tương ứng và gọi super đến lớp cha của nó để thực hiện các công việc mặc định của lớp trước khi viết lại cho hàm:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    super.prepare(for: segue, sender: sender)  
}
```

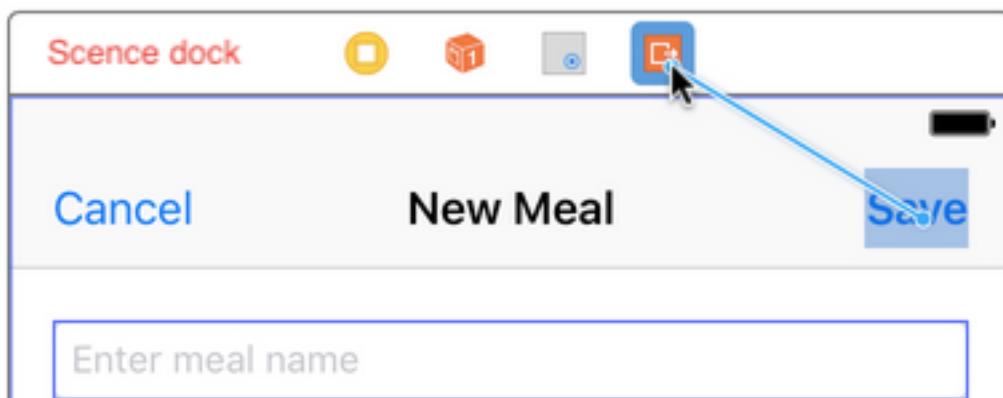
Việc chuẩn bị dữ liệu chỉ diễn ra khi nút Save được tap. Để có thể nhận biết được nút này, trước tiên liên kết code dạng tham chiếu với nút Save lấy tên là **btnSave** và ta dùng toán tử so sánh trùng (====) để kiểm tra (Xem lại chương 1). Nội dung hàm sẽ như sau:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    super.prepare(for: segue, sender: sender)  
    // Check if is btnSave?  
    if let button = sender as? UIBarButtonItem, button === btnSave {  
        let name = txtFoodName.text ?? ""  
        let image = mealImage.image  
        let rating = ratingControl.ratingValue  
        // Set new meal to pass to the Meal List Screen  
        meal = Meal(name: name, image: image, rating: rating)  
    }  
}
```

Việc di chuyển từ màn hình A sang màn hình B trong chế độ Modally Segue, muốn quay lại màn hình A từ B (Hoặc từ B ta có thể quay lại bất kỳ màn hình nào đã tồn tại trước đó chứ không tạo mới) thì cần các **Unwind Segue**. Mỗi khi một UnWindSegue được thực hiện nó sẽ thực hiện hàm prepare ở màn hình nguồn trước (các thao tác cần thiết trước khi chuyển màn hình) và sau khi thực hiện việc chuyển màn hình nó sẽ gọi đến một **unwind Action** (do lập trình viên tự định nghĩa và được kết nối với Unwind Segue) để thực hiện các thao tác cần thiết sau khi chuyển màn hình thành công. Unwind Action sẽ được định nghĩa ở màn hình đích (màn hình sẽ được chuyển đến A) theo một cú pháp nhất định và được kết nối với màn hình nguồn mỗi khi sự kiện chuyển màn hình xuất hiện. Trong ứng dụng này, chúng ta sẽ định nghĩa **Unwind Action** ở màn hình MealList (di chuyển quay về từ Meal Detail) như sau:

```
// Unwind to Meal List Screen
@IBAction func unWindToMealList(sender: UIStoryboardSegue) {
}
```

Đây là cú pháp bắt buộc của một **Unwind Action** với `@IBAction` ở đầu và một tham số dạng `UIStoryboardSegue`. Tên có thể đặt bất kỳ (ví dụ `unWindToMealList`). Việc còn lại là tạo ra một **Unwind Segue** mỗi khi nút Save trong màn hình `MealDetail` được tap vào và kết nối nó với **Unwind Action** đã được định nghĩa. Thực hiện `Ctrl + Drag` nút Save vào biểu tượng **Exit** trên **Scence dock** của màn hình `Meal Detail` (Hình 2.5.7.4).



Hình 2.5.7.4 Tạo Unwind Segue cho nút Save và kết nối với Unwind Action

Một bảng Action Segue xuất hiện với danh sách tất cả các **Unwind Action** đã được định nghĩa cho các màn hình (hiện tại chỉ có một Action Segue là `unWindToMealList`), hãy chọn `unWindToMealList` để kết nối **Unwind Segue** với **Unwind Action** đã được định nghĩa trong màn hình `MealList`. Chạy chương trình và cho nhận xét!

Khi quay lại màn hình `Meal List` từ màn hình `Meal Detail`, hàm `unWindToMealList` sẽ được gọi và các thao tác sau khi chuyển màn hình như: lấy món ăn mới truyền về, thêm món ăn mới vào danh sách món ăn và cập nhật trên Table View đều sẽ được thực hiện ở hàm này. Di chuyển tới hàm và sửa như sau:

```
// Unwind to Meal List Screen
@IBAction func unWindToMealList(sender: UIStoryboardSegue) {
    // Get the Source view controller in order to get the new meal
    if let sourceViewController = sender.source as? MealDetailController,
        let newMeal = sourceViewController.meal {
        // The new indexPath of the new meal in the table
        let newIndexPath = IndexPath(row: meals.count, section: 0)
        // Put the new meal into the datasource of table view
        meals += [newMeal]
        // Update the new meal in the table view
        tableView.insertRows(at: [newIndexPath], with: .automatic)
    }
}
```

Việc truyền tham số newMeal từ màn hình MealDetail về sẽ được lấy thông qua đối tượng segue của hàm (chính là sender), vì từ đối tượng này ta có thể lấy về đối tượng MealDetailController và qua đó truy xuất được biến meal (chứa giá trị new meal). Do việc lấy đối tượng MealDetailController cũng như lấy biến newMeal có thể trả về nil, nên ta cần unwrap nó bằng cấu trúc if ... let (Xem chương 1). Khi đã lấy được new meal từ bên MealDetail truyền về, thì việc cập nhật nó vào mảng datasource của Table view cũng như đưa new meal vào danh sách món ăn đã có là điều dễ dàng.

Bước 6: Hoàn thiện chức năng ứng dụng

Với ứng dụng hiện tại, nút Save có thể được tap bất cứ khi nào kể cả khi chưa đưa dữ liệu vào trong món ăn mới.

Bài tập: Hãy làm mờ nút Save (dùng lệnh `btnSave.isEnabled = false`) nếu như trường tên trong new meal đang rỗng hoặc khi bắt đầu soạn thảo cho trường tên.

Gợi ý: Nên viết một hàm **upDateSaveState** trong đó thay đổi trạng thái của **btnSave** theo trạng thái của **TextField**:

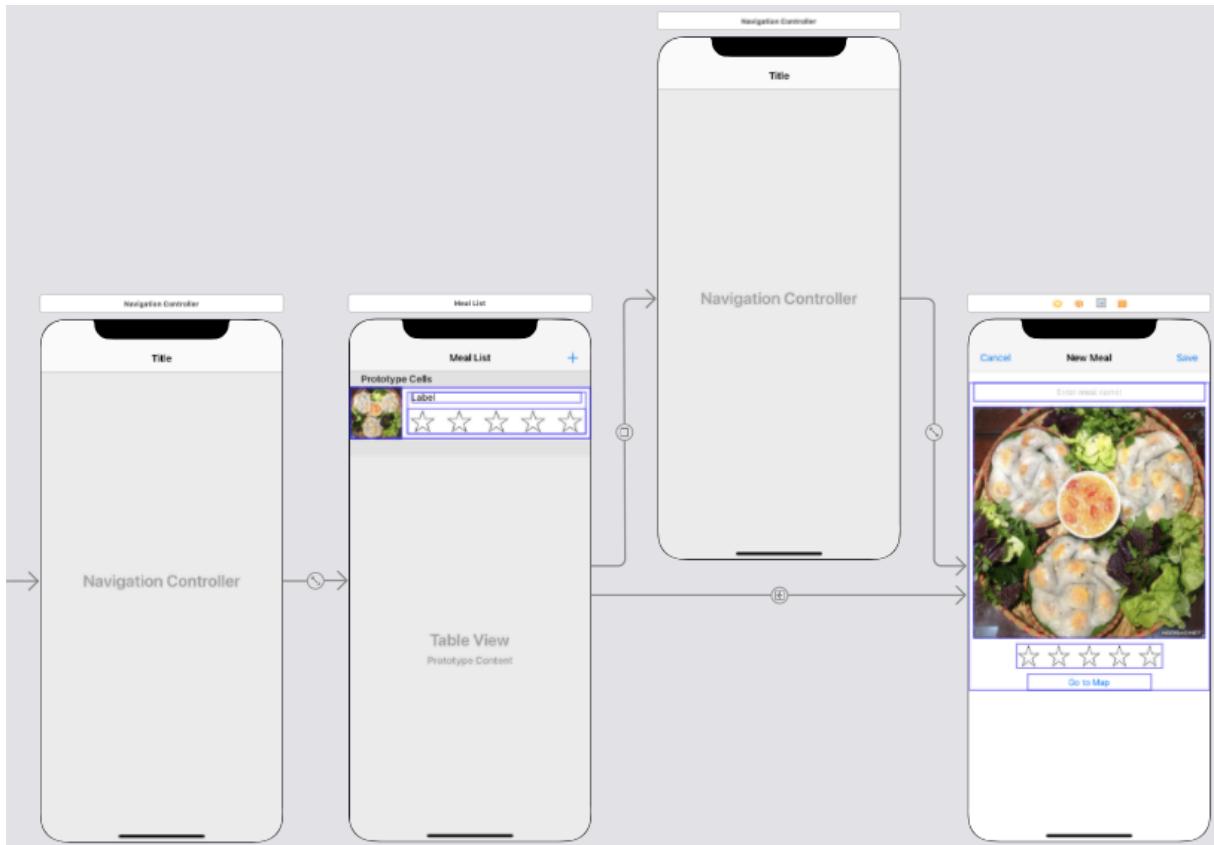
```
let name = txtFoodName.text ?? ""  
btnSave.isEnabled = !name.isEmpty
```

Và hàm này sẽ được gọi mỗi khi chuyển đến màn hình New meal (gọi trong `viewDidLoad`) cũng như mỗi khi kết thúc việc soạn thảo (gọi trong hàm `textFieldDidEndEditing`) để cập nhật lại trạng thái nút Save. Tuy nhiên, còn một trường hợp mỗi khi người dùng bắt đầu tap vào Textfield thì cũng phải disable nút Save (để tránh trường hợp người dùng xoá hết tên rồi tap luôn vào nút Save): Để thực hiện điều đó, cần định nghĩa thêm hàm uỷ quyền **textFieldDidBeginEditing** và trong đó thêm lệnh `btnSave.isEnabled = false`.

Ngoài ra, chúng ta cần thực hiện chức năng cho phép điều chỉnh, cập nhật nội dung của một món ăn có sẵn mỗi khi người sử dụng tap vào một dòng trên danh sách món ăn của màn hình Meal List. Như vậy, mỗi khi một dòng trong danh sách món ăn được tap ứng dụng cũng sẽ chuyển đến màn hình MealDetail nhưng với mục đích chỉnh sửa cho món ăn được tap lên. Do đó, việc di chuyển từ màn hình Meal List sang màn hình Meal Detail khác hoàn toàn với việc tạo mới một món ăn. Hãy thực hiện theo các bước như sau để tạo một “đường đi mới” từ MealList đến Meal Detail:

a) Ctrl + Drag từ Prototype Cell đến màn hình MealDetailController, chọn Show từ bảng Selection Segue Menu (Hình 2.5.7.5). Hãy đặt tên cho Segue mới: **editMealSegue**.

Bài tập: Chạy thử chương trình và cho nhận xét!



Hình 2.5.7.5 Hiện thực hoá chức năng cập nhật một món ăn đã có
b) Truyền tham số (thông tin món ăn sẽ được cập nhật) sang màn hình MealDetailController, cho phép người dùng có thể điều chỉnh món ăn đã có.

Bài tập: Hãy thực hiện việc truyền tham số từ màn hình MealList sang màn hình MealDetailController!

Gợi ý: Di chuyển đến phần //MARK: - Navigations, mở comment cho hàm **prepare** và thực hiện việc truyền tham số ở đó! Lưu ý, hàm này sẽ được gọi cả khi chúng ta di chuyển từ màn hình MealList sang màn hình MealDetail khi tap vào nút "+" hay tap lên một món ăn đã có. Do đó cần dùng switch hoặc if else để phân biệt các trường hợp này thông qua tên của mỗi loại segue đã được đặt trước đó (addNewMealSegue hay editMealSegue). Đoạn chương trình tham khảo có dạng:

```
if let segueName = segue.identifier {  
    switch segueName {
```

```

case "addNewMealSegue":
    break
case "editMealSegue":
    if let destinationController = segue.destination as? MealDetailController {
        if let selectedCell = sender as? MealTableViewCell {
            if let indexPath = tableView.indexPath(for: selectedCell) {
                destinationController.meal = meals[indexPath.row]
            }
        }
    }
default:
    print("See the names for these segues!")
}
else {
    print("The segue is not named!")
}

```

Bài tập: Hãy chuyển các cấu trúc if let lồng nhau thành cấu trúc guard và cho nhận xét!

c) Lấy tham số được truyền từ màn hình Meal List sang màn hình MealDetail.

Bài tập: Trong màn hình MealDetailController, hãy thực hiện lấy tham số truyền sang từ màn hình MealList mỗi khi người dùng tap vào một món ăn đã có!

Gợi ý: Vì biến meal là một biến Optional, nên hãy kiểm tra nó trong viewDidLoad, nếu có dữ liệu thì cập nhật lên giao diện!

Bài tập: Sau khi hoàn thiện, thực hiện chương trình và cho nhận xét!

d) Nhận dạng đường đi của việc chuyển màn hình. Do việc xử lý dữ liệu trong tạo món ăn mới và hiệu chỉnh món ăn đã có là khác nhau, đồng thời cơ chế hiển thị của màn hình MealDetailController cũng khác nhau (Modally và đầy màn hình chít tiết vào Navigation Stack) do đó cần có dấu hiệu nhận dạng. Đơn giản nhất, chúng ta sẽ dùng cấu trúc enum để ghi nhận dạng đường đi. Mặc định là newMeal:

```

enum NavigationType {
    case newMeal
    case editMeal
}
var navigationType: NavigationType = .newMeal

```

Khi đó, trong hàm prepare của màn hình MealListController chúng ta sẽ ghi nhận kiểu di chuyển tương ứng cho mỗi trường hợp.

Bài tập: Dựa vào cấu trúc enum và biến navigationType được định nghĩa cho màn hình MealDetailController, hãy sửa hàm prepare trong màn hình MealListController để nhận dạng đường đi của việc chuyển màn hình!

Bài tập: Dựa vào việc đã đánh dấu đường đi ở trên, điều chỉnh hàm unWindToMealList sao cho nếu thêm mới sẽ thực hiện như đã làm trước đó, nếu cập nhật món ăn thì thực hiện cập nhật trên datasource và trên table view (không thêm mới), đồng thời sửa lại hàm xử lý cho nút Cancel trong cả hai trường hợp.

Gợi ý: Có thể lấy indexPath của món ăn được tap trên màn hình MealListController bằng lệnh let selectedIndexPath = tableView.indexPathForSelectedRow. Còn xử lý nút Cancel trong trường hợp hiệu chỉnh món ăn thì cần dùng lệnh pop để lấy màn hình đó ra khỏi Navigation Stack (không thể dùng dismiss):

```
if let theNavigationController = navigationController {  
    theNavigationController.popViewController(animated: false)  
}
```

2.6 Câu hỏi và bài tập chương 2

1. Tại sao khi phát triển các ứng dụng trên di động cần chú trọng nhiều đến việc thiết kế giao diện chương trình (có thể tồn đến gần 50% công sức)?
2. Thực hiện mọi hoạt động và bài tập được yêu cầu trong toàn bộ chương 2!
3. Tìm hiểu về Closure trong Swift và cải tiến chương trình Calcultor!
4. Sửa lại ứng dụng Calculator sao cho sau khi thực hiện một phép toán, nếu nhấn vào phím 0 thì màn hình trở lại như ban đầu (Xoá kết quả, trở về màn hình 0) nhưng các trường hợp còn lại vẫn phải hoạt động như bình thường.
5. Để image view trong màn hình chi tiết của món ăn luôn hiển thị ảnh với cùng kích thước (không thay đổi theo kích thước của ảnh) thì phải làm gì? Hãy sửa và chạy thử! Sửa thuộc tính Content Mode của ImageView thành Scale to Fill. Chạy lại chương trình và cho nhận xét.
6. Thực hiện chương trình trong hoàn thiện trong 2.5.2, điều chỉnh xoá bỏ lựa chọn User Interaction Enable và cho biết kết quả? Giải thích!

7. Sau khi thực hiện chức năng trong 2.5.5, hãy chọn StackView của RatingControl, vào bảng thuộc tính điều chỉnh Distribution = Fill Equally. Quan sát sự thay đổi trên màn hình giao diện storyboard và cho nhận xét!
8. Tìm hiểu trong ứng dụng iOS, mỗi ảnh đưa vào ứng dụng có bao nhiêu trạng thái? Đó là những trạng thái nào ngoài 3 trạng thái đang sử dụng?
9. Trong mục 2.5.5 hướng dẫn cách đưa các thuộc tính vào bảng Attributes Inspector. Câu hỏi đặt ra là với những loại biến nào có thể đưa được vào đây? Với các kiểu dữ liệu tự tạo có thể đưa vào trong này không? Tại sao?
10. Nếu bước 5 trong mục 2.5.6 không có hai đoạn lệnh kiểm tra ràng buộc các thuộc tính thì vấn đề gì sẽ xảy ra? Giải thích!
11. Tìm hiểu về Table View nhiều Section trong iOS. Cải tiến màn hình MealViewController sao cho có thể hiển thị nhiều Sections khác nhau!
12. Trong bước 2, mục 2.5.7 khi thực hiện chức năng Edit và xoá một phần tử trong Table View thì giao diện của phần tử bị đẩy một phần ra ngoài màn hình. Hãy dựa trên những điều đã học về Autolayout, hiệu chỉnh giao diện cho Prototype Cell để không còn tình trạng đó xảy ra nữa.
13. Tại sao bước 5, mục 2.5.7 biến meal lại là biến Optional? Giải thích!
14. Hãy tìm hiểu và giải thích tại sao tại bước 4, mục 2.5.7 trong hàm prepare lại sử dụng câu lệnh: `let name = txtFoodName.text ?? ""`? Ý nghĩa của nó là gì?
15. * Hãy viết ứng dụng Quản lý nhân sự!
16. * Hãy viết ứng dụng trắc nghiệm khách quan với 4 dạng câu hỏi: Multi question multi-choices, multi-question one-choice, matching question và true-false question!
17. Phân biệt các dạng chuyển màn hình khác nhau trong iOS!
18. Viết ứng dụng Calculator (giống Android) sao cho việc chuyển màn hình sử dụng dạng Show segue! Hãy tìm hiểu và thực hiện truyền tham số giữa các màn hình để hoàn thiện ứng dụng!

CHƯƠNG 3. LÀM VIỆC VỚI CƠ SỞ DỮ LIỆU

Mục tiêu:

- Về kỹ năng:
 - + Thiết kế và xây dựng được cơ sở dữ liệu cho các ứng dụng trên iOS;
 - + Hiện thực hóa các ứng dụng vừa và nhỏ trên iOS;
- Về năng lực tự chủ và trách nhiệm:
 - + Luôn chủ động tìm hiểu vấn đề khi thực hiện các nhiệm vụ được giao;
 - + Luôn tuân thủ đầy đủ các quy định của lớp học.

Mô tả nội dung: Chương này sẽ hướng dẫn người học cách sử dụng Framework FMDB để thiết kế và cài đặt cơ sở dữ liệu SQLite cho các ứng dụng trên iOS.

3.1 Thiết kế Data model cho ứng dụng

3.1.1 Phân tích ứng dụng Quản lý món ăn

Như đã trình bày trong bước 5, mục 2.5.6, mỗi món ăn sẽ gồm 3 trường là Tên món ăn (kiểu chuỗi String), ảnh của món ăn (kiểu UIImage) và giá trị được đánh giá rating (kiểu Int và có khoảng giá trị từ 0 đến 5). Trong đó, trường tên không được phép rỗng; trường image có thể có hoặc không có dữ liệu (biến Optional) và trường rating bắt buộc giá trị phải nằm trong phạm vi cho phép.

3.1.2 Thiết kế Datamodel cho ứng dụng

Dựa trên những phân tích ở các mục trước, chúng ta xây dựng cấu trúc dữ liệu cho các món ăn như sau (Xem thêm bước 5, mục 2.5.6):

```
import UIKit
class Meal {
    //MARK: Properties
    var mealName: String
    var mealImage: UIImage?
    var ratingValue: Int
    //MARK: Initialization
    init?(name: String, image: UIImage?, rating: Int) {
        // Check the conditions
        guard !name.isEmpty else {
            return nil
        }
        guard (rating >= 0) && (rating <= 5) else {
            return nil
        }
    }
}
```

```

        // Initialization of class's properties
        mealName = name
        mealImage = image
        ratingValue = rating
    }
}

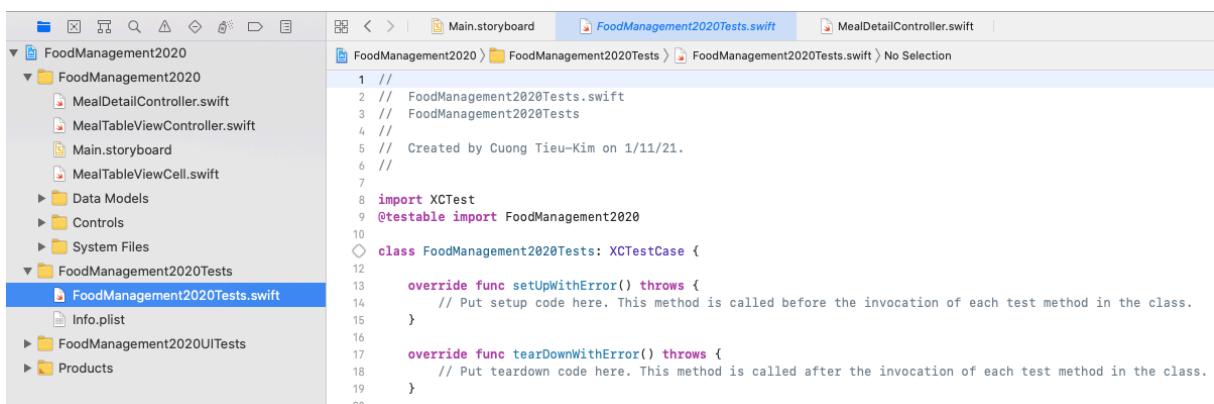
```

3.2 Kiểm thử tính đúng đắn của Data model

3.2.1 Xây dựng các test cases kiểm thử tính đúng đắn cho Datamodel của ứng dụng

Khi xây dựng các cấu trúc dữ liệu, luôn cần đảm bảo rằng chúng sẽ thỏa mãn mọi ràng buộc cần thiết để tránh mọi lỗi tiềm ẩn khi chạy chương trình. Để thực hiện điều đó, trong Xcode cho ta một công cụ để kiểm tra đó là XCTest framework. Hãy sử dụng XCTest framework để viết các Unit Test khi cần thiết trong suốt quá trình phát triển.

Hãy mở thư mục chứa các Unit Test cho chương trình Quản lý món ăn (Hình 3.2.1.1).



Hình 3.2.1.1 Mở file để viết các Unit Test trong iOS

Trong phạm vi giáo trình này chỉ hướng dẫn người học cách viết test case để kiểm thử tính đúng đắn của dữ liệu nên hãy xoá tất cả các hàm không cần thiết trong lớp đó. Giờ đây chúng ta sẽ viết tất cả các test cases cần thiết để kiểm tra tính đúng đắn của dữ liệu. Mỗi test case sẽ có cấu trúc giống hàm `testExample()` mới xoá. Bắt đầu của hàm bắt buộc là chữ **test** (nếu không hệ thống sẽ không hoạt động như mong muốn) và tiếp theo là tên của Unit test ta muốn xây dựng (Nguyên tắc viết các test cases như đã học trong môn học Kiểm thử phần mềm). Rõ ràng ở đây chúng ta cần hai hàm để kiểm tra mọi trường hợp thành công và mọi trường hợp thất bại khi tạo mới đối tượng món ăn và trong mỗi hàm cần xem xét đến mọi ràng buộc cũng như các điều kiện biên. Trường hợp thành công ta cần kiểm tra với tên bất kỳ không rỗng cùng giá trị rating nằm trong khoảng cho phép (chỉ cần kiểm tra hai giá trị biên là đủ). Trường hợp không thành công ta cần kiểm tra với giá trị rating nằm ngoài khoảng (-1 và 6 là đủ) và tên rỗng.

Lớp Unit test sẽ như sau:

```
class FoodManagement2020Tests: XCTestCase {
    // Test the Successes of Initialization of a meal
    func testMealInitializationSucceeds(){
        // Test Zero rating
        let zeroRatingMeal = Meal(name: "Zero rating meal", image: nil,
                                   rating: 0)
        XCTAssertNotNil(zeroRatingMeal)
        // Test the highest rating value
        let highestRatingMeal = Meal(name: "Highest rating", image:
                                       nil, rating: 5)
        XCTAssertNotNil(highestRatingMeal)
    }

    // Test the Fails of Initialization of a meal
    func testMealInitializationFails() {
        // Test negative rating value
        let negativeRatingMeal = Meal(name: "Negative rating", image:
                                       nil, rating: -1)
        XCTAssertNil(negativeRatingMeal)
        // Test over rating value
        let overRatingMeal = Meal(name: "Over rating meal", image: nil,
                                   rating: 6)
        XCTAssertNil(overRatingMeal)
        // Test empty name
        let emptyNameMeal = Meal(name: "", image: nil, rating: 4)
        XCTAssertNil(emptyNameMeal)
    }
}
```

3.2.2 Kiểm thử và điều chỉnh

Thực hiện chạy Unit Test bằng cách chọn: Product => Test, XCTest framework sẽ tiến hành thực hiện các unit test và cho kết quả mong muốn. Nếu pass sẽ có check màu xanh, nếu chưa đạt sẽ có check màu đỏ, khi đó cần điều chỉnh lại datamodel cho đến khi nào mọi unit test đều pass hết (Hình 3.2.2.1).

```
8 import XCTest
9 @testable import FoodManagement2020
10
11 class FoodManagement2020Tests: XCTestCase {
12     // Test the Successes of Initialization of a meal
13     func testMealInitializationSucceeds(){
14         // Test Zero rating
15         let zeroRatingMeal = Meal(name: "Zero rating meal", image: nil, rating: 0)
16         XCTAssertNotNil(zeroRatingMeal)
17         // Test the highest rating value
18         let highestRatingMeal = Meal(name: "Highest rating", image: nil, rating: 5)
19         XCTAssertNotNil(highestRatingMeal)
20     }

21     // Test the Fails of Initialization of a meal
22     func testMealInitializationFails() {
23         // Test negative rating value
24         let negativeRatingMeal = Meal(name: "Negative rating", image: nil, rating: -1)
25         XCTAssertNil(negativeRatingMeal)
26         // Test over rating value
27         let overRatingMeal = Meal(name: "Over rating meal", image: nil, rating: 6)
28         XCTAssertNil(overRatingMeal)
29         // Test empty name
30         let emptyNameMeal = Meal(name: "", image: nil, rating: 4)
31         XCTAssertNil(emptyNameMeal)
32     }
33 }
34 ]
```

Hình 3.2.2.1 Chỉ dừng lại khi mọi test cases đều pass hết

3.3 Một số dạng lưu trữ dữ liệu lâu dài trên ứng dụng iOS

3.3.1 Core Data

Core Data bản thân nó không phải là một cơ sở dữ liệu. Core Data là một Framework trong iOS cho phép lưu trữ và quản lý một mạng lưới các đối tượng. Bản thân Core Data không phải là một cơ sở dữ liệu nhưng nó có thể dùng SQLite để lưu trữ.

3.3.2 SQLite

SQLite là một hệ quản trị cơ sở dữ liệu nhúng cho các ứng dụng trên di động (Android, iOS, ...). Nó hoạt động không cần thông qua máy chủ mà giống như một thư viện được liên kết với các ứng dụng. Tuy nhiên, iOS không hỗ trợ Framework riêng cho hệ quản trị cơ sở dữ liệu SQLite giống như Core Data. Do đó chúng ta thường phải sử dụng các thư viện từ cộng đồng phát triển.

3.3.3 Lưu trữ trên mạng

Nếu dữ liệu được lưu trữ bằng Core Data hoặc SQLite, thì dữ liệu đó chỉ có tác dụng local cho chính máy điện thoại đang sử dụng. Nếu dữ liệu cần được truy xuất từ nhiều máy điện thoại khác nhau, dữ liệu cần được lưu trữ trên mạng hoặc trên đám mây (tên gọi là lưu trữ dữ liệu online). Với ứng dụng trên di động, có rất nhiều cách khác nhau để lưu trữ dữ liệu online. Một trong những công cụ hữu hiệu với nhà phát triển ứng dụng iOS hoặc Android đó là Firebase (Được phát triển bởi Google).

3.4 Cơ sở dữ liệu SQLite với các ứng dụng iOS

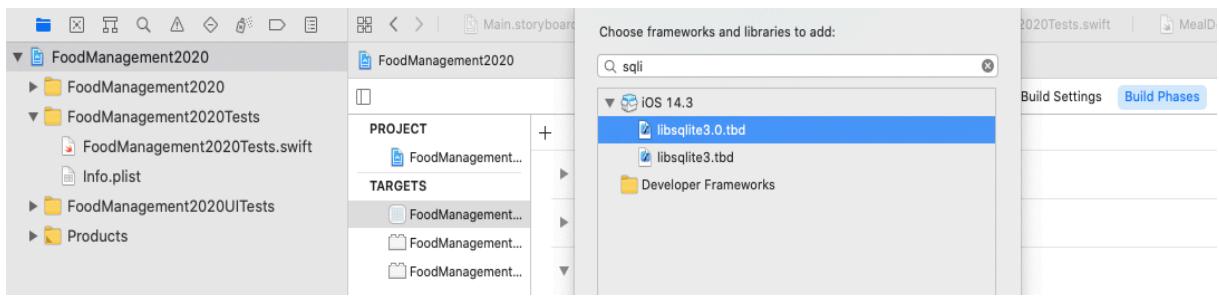
Để có thể phát triển các ứng dụng iOS sử dụng hệ quản trị cơ sở dữ liệu SQLite một cách hiệu quả, trong giáo trình này chúng tôi sẽ sử dụng thư viện FMDB (Flying Meat Database), đó là một thư viện viết bằng ngôn ngữ Objective-C nhưng có thể tích hợp vào các ứng dụng viết bằng Swift.

3.4.1 Cài đặt thư viện SQLite và Framework FMDB

Bước 1: Cài đặt thư viện SQLite. Trước tiên, cài đặt thư viện SQLite dynamic library (libsqlite3.0.tbd) cho ứng dụng bằng các thao tác sau:

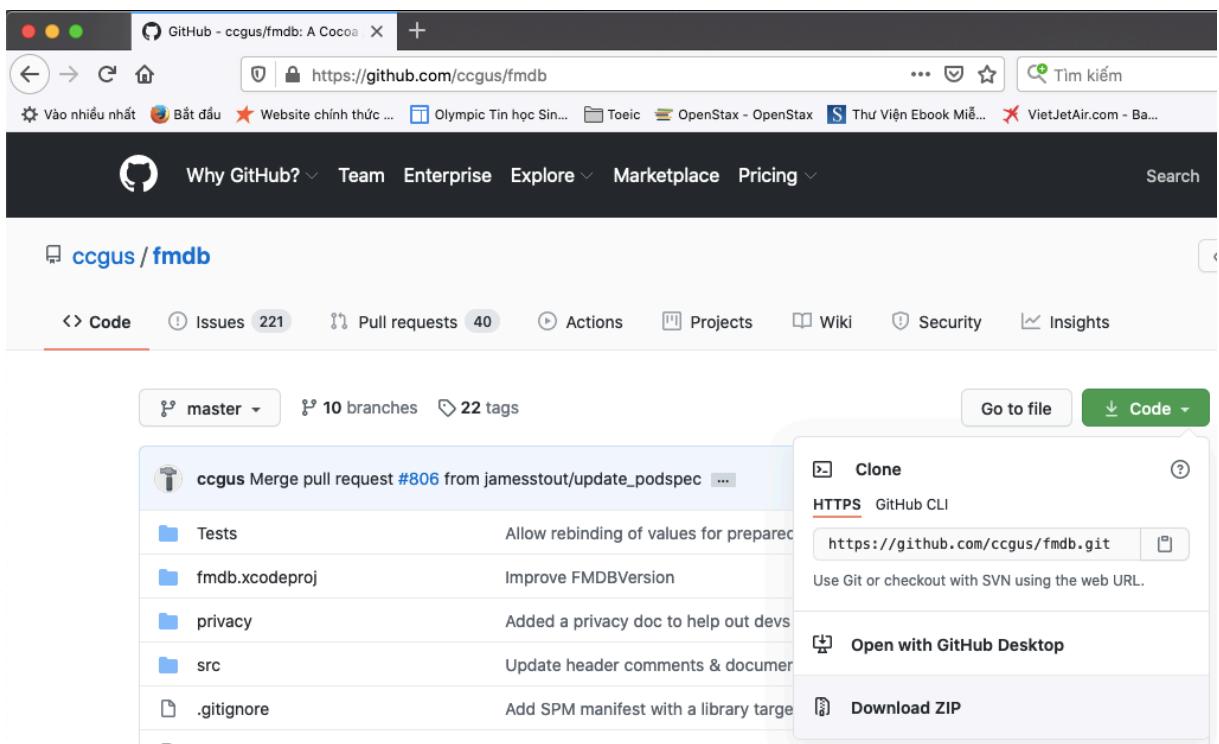
Thư mục gốc của Project => Build Phases => Link Binary with Libraries => (+) => Lựa chọn thư viện libsqlite30.tbd (Hình 3.4.1.1).

Bước 2: Download và cài đặt Framework FMDB cho ứng dụng.



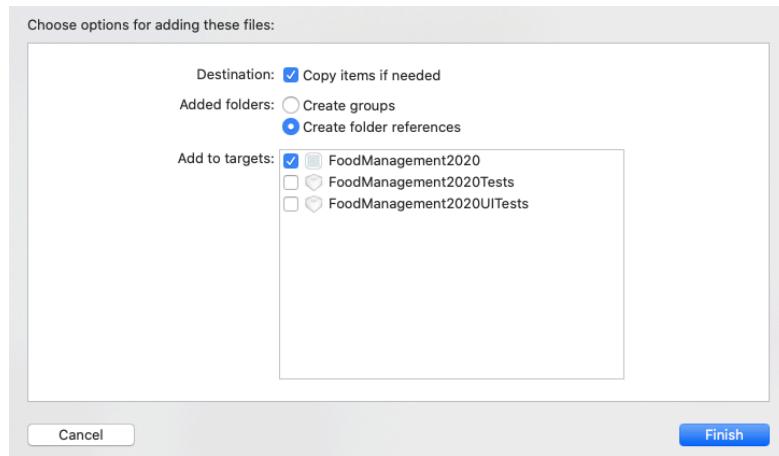
Hình 3.4.1.1 Cài đặt thư viện SQLite cho ứng dụng iOS

Có thể có nhiều cách cài đặt khác nhau như sử dụng chức năng checkout trong Source Control nếu ứng dụng đã cài đặt Git Repositories cho ứng dụng. Hoặc đơn giản nhất truy xuất vào trang Web: <https://github.com/ccgus/fmdb.git> và thực hiện download hoặc checkout code source về máy của mình (Hình 3.4.1.2).

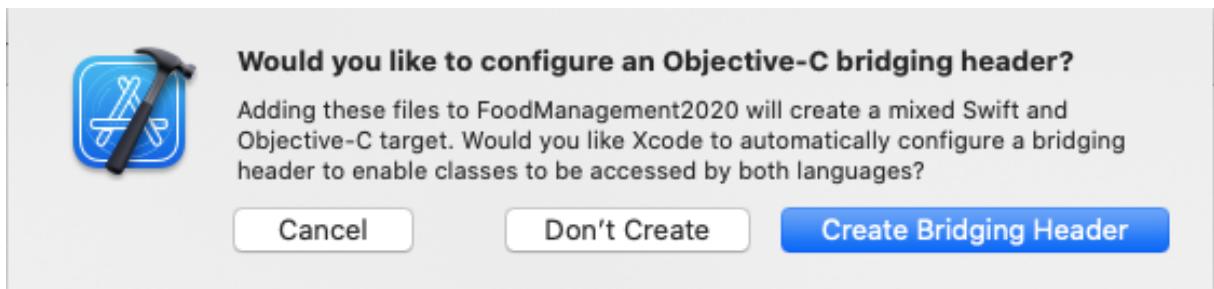


Hình 3.4.1.2 Download code source của FMDB

Tạo một thư mục có tên FMDB trong Project => Mở thư mục code mới download về “fmdb-master” => src => fmdb => Lựa chọn hết các files trong đó (trừ ra info.plist) => Kéo thả vào thư mục FMDB vừa tạo cho Project => Check chọn “Copy items if needed” trong mục Destination; chọn Create folder references cho mục Added folders; còn lại để mặc định (Hình 3.4.1.3) => Một Dialog khác xuất hiện => Chọn “Create Bridging Header” để tạo file header cho việc sử dụng thư viện này với Project dùng ngôn ngữ Swift (Hình 3.4.1.4).



Hình 3.4.1.3 Tuỳ chọn tích hợp thư viện FMDB vào Project



Hình 3.4.1.4 Tạo cầu nối thư viện Objective-C với ngôn ngữ Swift

File cầu nối này sẽ có tên trùng với tên Project. Tìm và mở file FoodManagement2020-Bridging-Header.h và thêm vào dòng lệnh sau vào file đó và thư viện đã sẵn sàng sử dụng trong Swift project hiện tại:

```
#import "FMDB.h"
```

3.4.2 Thiết kế tầng truy xuất dữ liệu DAL

DAL là viết tắt của Database Access Layer nhằm tạo ra một tầng riêng biệt về cơ sở dữ liệu cho các ứng dụng iOS, giúp cho các ứng dụng iOS dễ thiết kế và dễ bảo trì hơn. Ý tưởng của tầng truy xuất dữ liệu DAL là xây dựng một hệ thống các chức năng cơ bản của mọi cơ sở dữ liệu như đóng, mở, tạo bảng,... những chức năng này được gọi là các Database Primitives (có thể hiểu chúng như những viên gạch của tầng cơ sở dữ liệu). Từ những Database Primitives này chúng ta sẽ xây dựng nên một hệ thống các APIs sẽ được gọi ở tầng trên (để đảm bảo tính độc lập của tầng trên với những tầng với tầng DAL).

Dựa trên những thiết kế như trên, chúng ta sẽ tạo một Group riêng có tên là DAL để phục vụ cho việc tổ chức và thiết kế các Database Primitives và các Database APIs dựa trên **Framework FMDB**. Việc thiết kế tầng dữ liệu này sẽ gắn liền với ứng dụng

Quản lý món ăn đang thực hiện như một Case Study cho người học dễ hiểu hơn. Trong thực tế, có thể các Database Primitives và các Database APIs phức tạp hơn nhiều, tuy nhiên chúng ta vẫn hoàn toàn có thể mở rộng tầng DAL sau này mà không ảnh hưởng nhiều đến các chức năng đã viết ở tầng trên (do tính độc lập của chúng). Trong giáo trình này, chúng tôi thiết kế các Database Primitives và các Database APIs trong cùng một file swift có tên **FoodManagementDatabase**:

```
import Foundation
import UIKit
import os.log

class FoodManagementDatabase {
```

}

3.5 Xây dựng tầng truy xuất dữ liệu cho ứng dụng iOS

Trước tiên cần định nghĩa các thuộc tính của cơ sở dữ liệu như tên, đường dẫn, các thuộc tính của bảng dữ liệu... Đó sẽ là những thuộc tính được sử dụng bên trong tầng DAL (tầng trên sẽ không quan tâm đến những thuộc tính đó). Với ứng dụng Quản lý món ăn, thì chúng ta chỉ cần một bảng duy nhất gồm các trường `_id` (khoá nguyên, tăng tự động), `name` (kiểu text), `image` (lưu dưới dạng text) và `rating` (kiểu Int):

```
//MARK: Database Properties
let dPath: String
let DB_NAME: String = "Foods.sqlite"
let db: FMDatabase?

//MARK: Tables's properties
let TABLE_NAME: String = "meals"
let TABLE_ID: String = "_id"
let MEAL_NAME: String = "name"
let MEAL_IMAGE: String = "image"
let MEAL_RATING: String = "rating"
```

3.5.1 Thiết kế các chức năng cơ bản: Đóng, mở, tạo bảng...

Trong giáo trình này, chúng ta sẽ xây dựng những primitives sau cho cơ sở dữ liệu của chúng ta: hàm khởi tạo cho cơ sở dữ liệu, mở cơ sở dữ liệu, đóng cơ sở dữ liệu và hàm tạo bảng dữ liệu:

```
//MARK: Database Primitives
init() {
    let directories:[String] =
        NSSearchPathForDirectoriesInDomains(.documentDirectory,
        .allDomainsMask, true)
    dPath = directories[0] + "/" + DB_NAME
```

```

db = FMDatabase(path: dPath)
if db == nil {
    os_log("Can not create the database. Please review the dPath!")
}
else {
    os_log("Database is created successful!")
}
}

// Create table
func createTable() -> Bool {
    var ok: Bool = false
    if db != nil {
        let sql = "CREATE TABLE " + TABLE_NAME + "(" +
            + TABLE_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            + MEAL_NAME + " TEXT, " +
            + MEAL_IMAGE + " TEXT, " +
            + MEAL_RATING + " INTEGER)"

        if db!.executeStatements(sql) {
            ok = true
            os_log("Table is created!")
        }
        else {
            os_log("Can not create the table!")
        }
    }
    else {
        os_log("Database is nil!")
    }
    return ok
}
// Open database
func open() -> Bool {
    var ok: Bool = false
    if db != nil {
        if db!.open() {
            ok = true
            os_log("The database is opened!")
        }
        else {
            print("Can not open the Database:
                \n\((db!.lastErrorMessage())"))
        }
    }
    else {
        os_log("Database is nil!")
    }
    return ok
}
// Close database
func close(){
    if db != nil {
        db!.close()
    }
}

```

3.5.2 Thiết kế các API cho tầng trên

Database APIs thực chất là các hàm cần thiết để tầng trên có thể gọi xuống cơ sở dữ liệu mà không cần quan tâm đó là cơ sở dữ liệu gì và làm sao thực hiện được chúng. Trong ứng dụng Quản lý món ăn, rõ ràng chúng ta cần các APIs cho phép đọc từ cơ sở dữ liệu và đưa kết quả vào danh sách món ăn, ghi từng món ăn hoặc danh sách các món ăn vào cơ sở dữ liệu, cập nhật một món ăn vào cơ sở dữ liệu, xoá bỏ một món ăn từ cơ sở dữ liệu. Ngoài ra, để mở rộng cho khả năng của cơ sở dữ liệu như tránh ghi nhiều món ăn trùng nhau vào cơ sở dữ liệu,... chúng ta cũng cần một API phục vụ cho việc tìm kiếm sự tồn tại của một món ăn trong cơ sở dữ liệu.

a. Đọc dữ liệu từ cơ sở dữ liệu

Với ứng dụng Quản lý món ăn, mỗi khi chạy chương trình ta cần kiểm tra xem trong cơ sở dữ liệu có lưu trữ danh sách món ăn chưa? Nếu có rồi thì cần đọc toàn bộ bảng dữ liệu trong cơ sở dữ liệu và đưa vào mảng dữ liệu món ăn (chính là datasource của Table View). Do đó, tham số của hàm sẽ được thiết kế với từ khoá **inout** (Xem chương 1):

```
func readMealsList(meals:&inout [Meal]) {
    if db != nil {
        var results: FMResultSet?
        let sql = "SELECT * FROM \(TABLE_NAME)"
        // Query
        do {
            results = try db!.executeQuery(sql, values: nil)
        }
        catch {
            print("Fail to read data: \(error.localizedDescription)")
        }
        // Read data from the results
        if results != nil {
            while (results?.next())! {
                let mealName = results!.string(forColumn: MEAL_NAME)
                let stringImage = results!.string(forColumn: MEAL_IMAGE)
                let mealRating = results!.int(forColumn: MEAL_RATING)
                // Transform string image to UIImage
                let dataImage: Data = Data(base64Encoded: stringImage!,
                                            options: .ignoreUnknownCharacters)!
                let mealImage = UIImage(data: dataImage)
                // Create a meal to contain the values
                let meal = Meal(name: mealName!, image: mealImage!,
                                rating: Int(mealRating))
                meals.append(meal!)
            }
        }
    } else{
        os_log("Database is nil!")
    }
}
```

b. Ghi dữ liệu vào cơ sở dữ liệu

Ứng dụng Quản lý món ăn cần một hàm sao cho mỗi lần gọi nó có thể ghi một dữ liệu kiểu món ăn vào cơ sở dữ liệu, do đó nó có một tham biến kiểu món ăn:

```
func insertMeal(meal: Meal){  
    if db != nil{  
        // Transform the meal image to String  
        let imageData: NSData = meal.mealImage!.pngData()! as NSData  
        let mealImageString = imageData.base64EncodedData(options:  
            .lineLength64Characters)  
        let sql = "INSERT INTO " + TABLE_NAME + "(" + MEAL_NAME + ", "  
            + MEAL_IMAGE + ", " + MEAL_RATING + ")" + " VALUES (?, ?, ?)"  
        if db!.executeUpdate(sql, withArgumentsIn: [meal.mealName,  
            mealImageString, meal.ratingValue]) {  
            os_log("The meal is insert to the database!")  
        }  
        else {  
            os_log("Fail to insert the meal!")  
        }  
    }  
    else {  
        os_log("Database is nil!")  
    }  
}
```

c. Cập nhật dữ liệu vào cơ sở dữ liệu

Mỗi khi chúng ta điều chỉnh một món ăn, rõ ràng ta cần cập nhật lại món ăn đó vào cơ sở dữ liệu. Như vậy, hàm cập nhật sẽ cần món ăn mới (để ghi vào cơ sở dữ liệu), món ăn cũ (để tìm kiếm trong cơ sở dữ liệu đúng chỗ cần cập nhật):

```
func updateMeal(oldMeal: Meal, newMeal: Meal){  
    if db != nil {  
        let sql = "UPDATE \(TABLE_NAME) SET \(MEAL_NAME) = ?,  
            \(MEAL_IMAGE) = ?, \(MEAL_RATING) = ? WHERE \(MEAL_NAME) = ? AND  
            \(MEAL_RATING) = ?"  
        // Transform image of new meal to String  
        let newImageData: NSData = newMeal.mealImage!.pngData()! as NSData  
        let newStringImage = newImageData.base64EncodedString(options:  
            .lineLength64Characters)  
        // Try to query the database  
        do{  
            try db!.executeUpdate(sql, values: [newMeal.mealName,  
                newStringImage, newMeal.ratingValue, oldMeal.mealName,  
                oldMeal.ratingValue])  
            os_log("Successful to update the meal!")  
        }  
        catch{  
            print("Fail to update the meal:  
                \(error.localizedDescription)")  
        }  
    }  
    else {  
        os_log("Database is nil!")  
    }  
}
```

```
        }
    }
```

d. Xoá dữ liệu từ cơ sở dữ liệu

Mỗi khi chúng ta xoá một món ăn trong danh sách món ăn, rõ ràng ta cũng cần xoá nó khỏi cơ sở dữ liệu, do đó hàm chỉ cần một tham số là món ăn cần xoá:

```
func deleteMeal(meal: Meal){
    if db != nil {
        let sql = "DELETE FROM \(TABLE_NAME) WHERE \(MEAL_NAME) = ? AND
                  \(MEAL_RATING) = ?"
        do {
            try db!.executeUpdate(sql, values: [meal.mealName,
                                                meal.ratingValue])
            os_log("The meal is deleted!")
        } catch {
            os_log("Fail to delete the meal!")
        }
    } else {
        os_log("Database is nil!")
    }
}
```

e. Tìm kiếm dữ liệu từ cơ sở dữ liệu

Bài tập: Dựa vào hàm đọc dữ liệu, hàm cập nhật và hàm xoá dữ liệu hãy tự viết API cho việc tìm kiếm một món ăn trong cơ sở dữ liệu. Hàm sẽ có một biến truyền vào chính là món ăn cần tìm, nếu tìm thấy trong cơ sở dữ liệu có món ăn trùng tên và trùng rating thì trả về kết quả true, ngược lại trả về false.

3.6 Sử dụng tầng DAL cho ứng dụng iOS

Sau khi thiết kế xong tầng DAL, chúng ta có thể sử dụng chúng ở tầng trên cho việc lưu trữ dữ liệu của ứng dụng Quản lý món ăn. Rõ ràng mọi thao tác liên quan đến danh sách món ăn đều nằm ở màn hình MealListController. Nên việc sử dụng các Database APIs cũng nằm ở lớp này. Mở file MealListController, khai báo hai biến sau:

```
private var dao = FoodManagementDatabase()
static private var tableCreated: Bool = false
```

Trong đó biến **dao** (Viết tắt của Database Access Object) là một đối tượng của cơ sở dữ liệu cho phép chúng ta có thể sử dụng các Database APIs để thực hiện các thao tác đọc, ghi, cập nhật, tìm kiếm, xoá... các món ăn từ/vào/trong cơ sở dữ liệu SQLite của ứng dụng. Còn biến **tableCreated** để đảm bảo rằng không tạo lại bảng dữ liệu nếu nó đã có.

Di chuyển đến hàm viewDidLoad và điều chỉnh đoạn chương trình thêm dữ liệu giả cho Table view nhằm đảm bảo rằng nếu dữ liệu chưa có thì khởi tạo bảng dữ liệu, đọc dữ liệu từ cơ sở dữ liệu vào datasource của table view (nếu có), nếu không có thì vẫn tạo dữ liệu món ăn giả và ghi luôn vào cơ sở dữ liệu:

```
//Load from database
if dao.open() {
    if !MealTableViewController.tableCreated {
        MealTableViewController.tableCreated = dao.createTable()
    }
    dao.readMeals(meals: &meals)
}
if meals.count == 0 {
    // Create an example of meal
    let image = UIImage(named: "default")
    if let meal = Meal(name: "Mon Hue", image: image, rating: 3) {
        meals += [meal]
        if dao.open() {
            dao.insertMeal(meal: meal)
        }
    }
}
```

Di chuyển đến hàm thực hiện xoá món ăn trong menu Edit và điều chỉnh thành:

```
// Delete the meal from database
if dao.open() {
    dao.deleteMeal(meal: meals[indexPath.row])
}
// Delete the row from the data source
meals.remove(at: indexPath.row)
tableView.deleteRows(at: [indexPath], with: .fade)
```

Mỗi khi một món ăn bị xoá khỏi table view thì nó cũng được xoá khỏi cơ sở dữ liệu.

Bài tập: Tại sao lại cần thực hiện xoá trong cơ sở dữ liệu trước khi xoá trong datasource và trong table view?

Di chuyển đến hàm unWindToMealsList và điều chỉnh thành:

```
if sourceViewController.navigationItem.rightBarButtonItem?.title == "Add" {
    // Add the new meal to the database
    if dao.open() {
        let newMeal = Meal(name: "New Meal", image: nil, rating: 0)
        dao.insertMeal(meal: newMeal)
    }
}
// Update the table view
tableView.reloadData()
```

```

// For edit meal segue
else if sourceViewController.navigationController.navigationType == .editMeal{
    if let selectedIndexPath = tableView.indexPathForSelectedRow {
        // Update in the database
        if dao.open() {
            dao.updateMeal(oldMeal: meals[selectedIndexPath.row],
                           newMeal: newMeal)
        }
        // Update in datasource array
        meals[selectedIndexPath.row] = newMeal
        // Update in table view
        tableView.reloadRows(at: [selectedIndexPath], with: .none)
    }
}

```

Bài tập: Tại sao trong phần edit meal lại cần ghi thông tin vào cơ sở dữ liệu trước? Trong phần thêm một món ăn mới nếu ghi thông tin món ăn mới vào cơ sở dữ liệu trước có được không? Nếu cải tiến đưa thêm vào API tìm kiếm món ăn để tránh ghi trùng lặp trong cơ sở dữ liệu thì đoạn chương trình ghi vào cơ sở dữ liệu nên đặt ở đâu?

3.7 Câu hỏi và bài tập chương 3

1. Thiết kế lại cấu trúc của bảng dữ liệu nếu ta muốn lưu các món ăn dưới dạng tên, đường dẫn đến file ảnh, giá trị đánh giá! Tìm hiểu và viết lại chương trình!
2. Thực hiện mọi yêu cầu bài tập trong toàn bộ chương 3!
3. Khi nào dùng SQLite hiệu quả hơn? Khi nào dùng Core Data? Khi nào dùng Firebase?
4. Sửa lại hàm **init** sao cho trước khi khởi tạo cơ sở dữ liệu cần kiểm tra sự tồn tại của file chứa database trước?
5. Viết Database API mới dùng để tìm kiếm một món ăn trong cơ sở dữ liệu theo tên món ăn và trả về một danh sách các món ăn trùng tên hoặc nil?
6. Viết Database API mới dùng để tìm kiếm những món ăn có rating lớn hơn (nhỏ hơn) món ăn nào đó, hàm trả về danh sách các món ăn thoả điều kiện hoặc nil?

CHƯƠNG 4. BẢN ĐỒ TRỰC TUYẾN TRÊN IOS

Mục tiêu:

- Về kiến thức:
 - + Vận dụng ngôn ngữ Swift phát triển các ứng dụng trên iOS;
- Về kỹ năng:
 - + Xây dựng được các ứng dụng vừa và nhỏ sử dụng Webservice và các dịch vụ cơ bản (Map và Location);
 - + Hiện thực hóa các ứng dụng vừa và nhỏ trên iOS;
- Về năng lực tự chủ và trách nhiệm:
 - + Luôn chủ động tìm hiểu vấn đề khi thực hiện các nhiệm vụ được giao;
 - + Luôn tuân thủ đầy đủ các quy định của lớp học.

Mô tả nội dung: Chương này sẽ trình bày các kiến thức, kỹ năng cần thiết trong việc sử dụng các dịch vụ tiện ích (Map và Location) vào phát triển các ứng dụng iOS.

4.1 Một số dạng bản đồ trực tuyến thông dụng

Khác với các ứng dụng trên Android, với các ứng dụng trên iOS chúng ta có nhiều lựa chọn hơn khi dùng bản đồ trực tuyến và xác định vị trí (Map and Location). Ngoài những dịch vụ hỗ trợ rất mạnh từ Google ra, bản thân Apple cũng tự phát triển riêng các dịch vụ của mình và có tính năng gần như tương đương nhưng dễ sử dụng hơn (Map Kit).

4.1.1 Map Kit

Là dịch vụ Map and Location do Apple tự phát triển nhằm cung cấp các tính năng như hiển thị bản đồ trực tuyến, tương tác với bản đồ trực tuyến, định vị và tìm kiếm vị trí trên bản đồ trực tuyến, chỉ đường trên bản đồ trực tuyến ... cho các ứng dụng trên iOS. MapKit tiếp cận dựa trên việc xây dựng các tuyến đường (Khác với Google Map). Trước đây, Map Kit còn nhiều tính năng không được hỗ trợ tốt như Google Map, tuy nhiên cho tới thời điểm hiện tại, Map Kit đã hoàn toàn có thể thay thế cho Google Map và được sử dụng rộng rãi với hầu hết các ứng dụng trên iOS do giao diện đẹp và dễ sử dụng của nó (do nó đã được tích hợp sẵn trong các thư viện của Xcode). Ngoài ra, nó còn hỗ trợ chương trình Siri tốt hơn và hiệu quả hơn OK Google.

4.1.2 Google Map

Là dịch vụ bản đồ trực tuyến do Google phát triển cung cấp đầy đủ các tính năng cần thiết cho các ứng dụng trên Android và iOS với tiếp cận dựa trên Location và Places. Tuy nhiên, để có thể sử dụng chúng trong các ứng dụng iOS, chúng ta cần tích hợp chúng vào trong Project iOS. Điều này thường gây khó khăn cho các lập trình viên.

4.2 Sử dụng bản đồ trực tuyến trong các ứng dụng iOS

Trong giáo trình này, chúng tôi chỉ tập trung hướng dẫn người học sử dụng các dịch vụ Map and Location của Map Kit vào phát triển ứng dụng trên iOS.

4.2.1 Tích hợp MapKit vào ứng dụng

Để tích hợp MapKit vào ứng dụng iOS, chỉ cần thực hiện một câu lệnh đơn giản để import thư viện MapKit vào trong ứng dụng (Với Google map thì phức tạp hơn nhiều). Nếu muốn sử dụng các dịch vụ về Location thì cũng cần import thư viện tương ứng:

```
import MapKit  
import CoreLocation
```

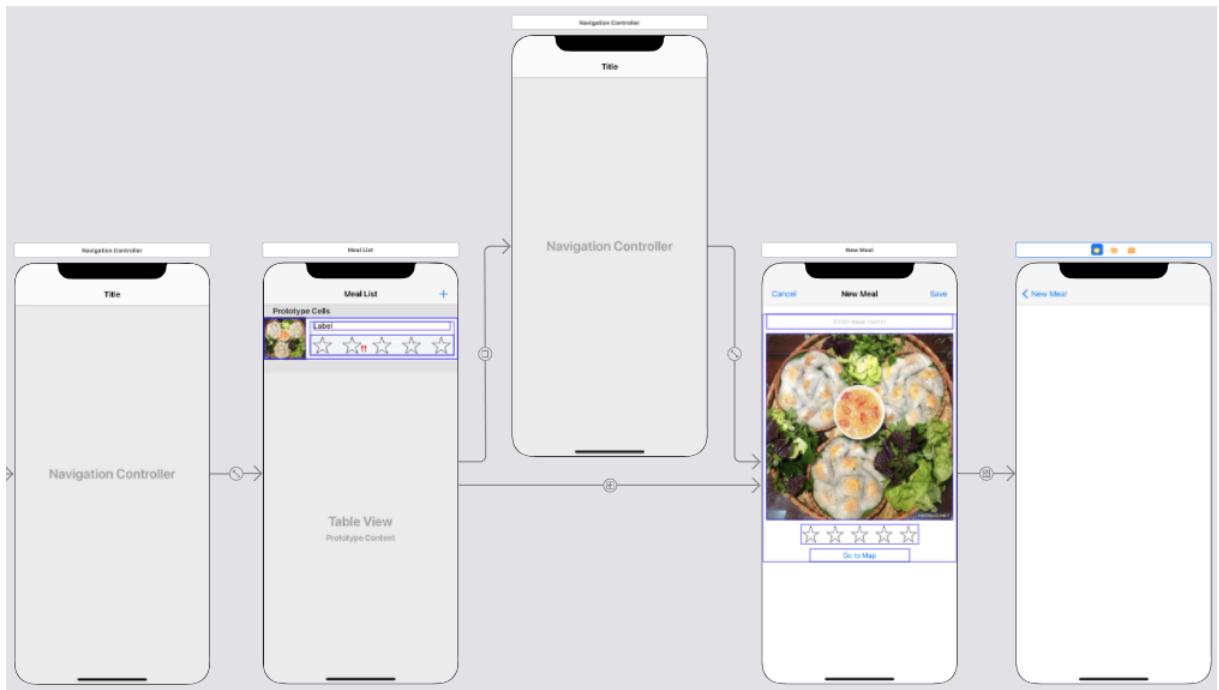
4.2.2 Tạo mới bản đồ trong ứng dụng iOS

Để sử dụng bản đồ trực tuyến trên iOS, chúng ta chỉ cần thực hiện hai thao tác cơ bản đó là: Khai báo bản đồ (bằng lệnh var mapView: MKMapView!) và khởi tạo cho bản đồ trực tuyến đó (init và định nghĩa lại hàm viewDidLoadSubview).

Bước 1: Tạo màn hình MapViewController mới.

Với ứng dụng của chúng ta, mỗi khi người dùng tap vào nút “Go to Map” sẽ hiển thị bản đồ trực tuyến và thực hiện những chức năng cơ bản như đánh dấu, tìm đường... Như vậy, chúng ta cần một màn hình mới có tên MapViewCotroller: Kéo thả một View Controller mới từ thư viện đối tượng vào Storyboard. Sắp xếp màn hình mới nằm ngay cạnh bên phải màn hình MealDetailController. Tiếp theo Ctrol + Drag từ nút “Go to Map” trong màn hình MealDetailController sang màn hình mới tạo => Chọn Show (Kết quả như hình 4.2.2.1).

Tiếp theo, cần tạo một lớp MapViewController.swift, kế thừa từ lớp UIViewController và liên kết nó với màn hình mới vừa tạo. Mở file MapViewController.swift để khai báo bản đồ trực tuyến và khởi tạo cho nó.



Hình 4.2.2.1 Tạo màn hình MapViewController

Bước 2: Tích hợp MapKit, CoreLocation và khởi tạo cho bản đồ trực tuyến.

Lớp MapViewController ban đầu có dạng như sau:

```

import UIKit
import MapKit
import CoreLocation

class MapViewController: UIViewController {
    //MARK: Properties
    var mapView: MKMapView!

    override func viewDidLoad() {
        super.viewDidLoad()
        //MARK: Initialization for Map
        mapView = MKMapView.init()
        view.addSubview(mapView)
    }

    //MARK: For Map
    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()
        mapView.frame = CGRect.init(origin: CGPoint.zero, size:
            view.frame.size)
    }
}

```

Do ban đầu lúc khởi tạo bản đồ trực tuyến chưa có kích thước (chiều dài, chiều rộng), nên nếu không khởi tạo kích thước cho bản đồ trong hàm `viewDidLayoutSubviews` thì chúng ta sẽ không nhìn thấy bản đồ khi chạy chương trình. Ở đây chúng ta cho bản đồ từ toạ độ 0 (góc trên bên trái màn hình) đến toạ độ max (góc dưới bên phải màn hình).

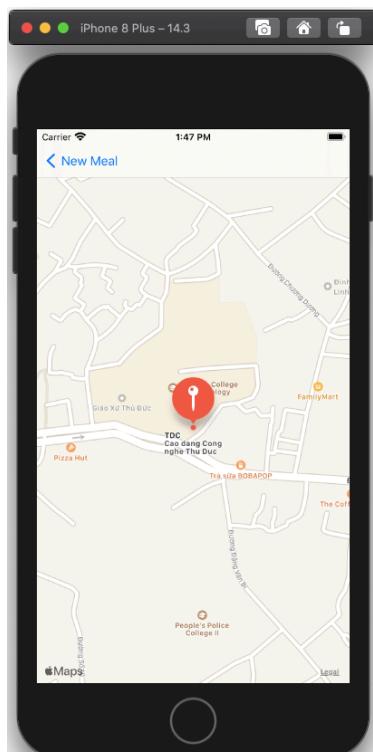
Bài tập: Chạy thử chương trình di chuyển đến màn hình MapViewController và cho nhận xét!

4.2.3 Đánh dấu trên bản đồ dùng Annotation

Trên bản đồ online nhiều khi ta cần đánh dấu (Marker) lại một số điểm cho nhiều mục đích khác nhau, mỗi điểm có những thông tin như tọa độ (kinh độ, vĩ độ), tên, hình ảnh,... Để có thể đánh dấu được trên bản đồ online trước tiên ta cần định nghĩa lớp Map annotation có tên MyMapAnnotation như sau:

```
import UIKit
import MapKit

class MapAnnotation: NSObject, MKAnnotation {
    //MARK: MKAnnotation protocol
    public var coordinate: CLLocationCoordinate2D
    public var title: String?
    public var subtitle: String?
    //MARK: Initializer
    init(coordinate: CLLocationCoordinate2D, title: String?, subtitle: String?) {
        self.coordinate = coordinate
        self.title = title
        self.subtitle = subtitle
        super.init()
    }
}
```



Trong hàm viewDidLoad hãy thực hiện những công việc sau để đánh dấu tọa độ của trường Cao đẳng Công nghệ Thủ Đức trên bản đồ:

```
// Add a marker on the Map
let tdc = CLLocation(latitude: 10.851261907187268, longitude: 106.75824763345204)
let tdcMarker = MyMapAnnotation(coordinate: tdc.coordinate, title: "TDC", subtitle: "Cao dang Cong nghe Thu Duc")
mapView.addAnnotation(tdcMarker)
```

Nếu sử dụng kết quả bài tập số 1 của chương này, chạy chương trình ta sẽ có kết quả như hình 4.2.3.1 trong đó biến tdc chứa tọa độ (kinh độ, vĩ độ) của trường Cao đẳng Công nghệ Thủ Đức, phạm vi nhìn là 500 m.

Hình 4.2.3.1 Đánh dấu trên bản đồ

4.2.4 Lấy vị trí hiện tại của người dùng trên bản đồ

Để có thể lấy được vị trí hiện tại của người dùng trên bản đồ, cần thực thi các hàm cần thiết trong `CLLocationManagerDelegate` đồng thời cần cho phép truy xuất vào vị trí hiện tại của người dùng. Để yêu cầu cấp quyền truy xuất vị trí hiện tại, mở file info.plist, rồi thêm vào Key: “Privacy - Location When In Use Usage Description” với thông báo “Allow to access user's Location!” (Hình 4.2.4.1).

FoodManagement2020 > FoodManagement2020 > System Files > Info.plist > No Selection		
Key	Type	Value
▼ Information Property List	Dictionary (17 items)	
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	\$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
► Application Scene Manifest	Dictionary (2 items)	
Application supports indirect input events	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array (1 item)	
► Supported interface orientations	Array (3 items)	
► Supported interface orientations (iPad)	Array (4 items)	
Privacy - Location When In Use Usage Description	String	Allow to access user's Location!

Hình 4.2.4.1 Cấp quyền cho việc truy xuất vị trí hiện tại trên bản đồ

Cần khai báo hai biến để chứa tọa độ ví trí hiện tại và biến quản lý việc truy xuất vị trí hiện tại trên bản đồ:

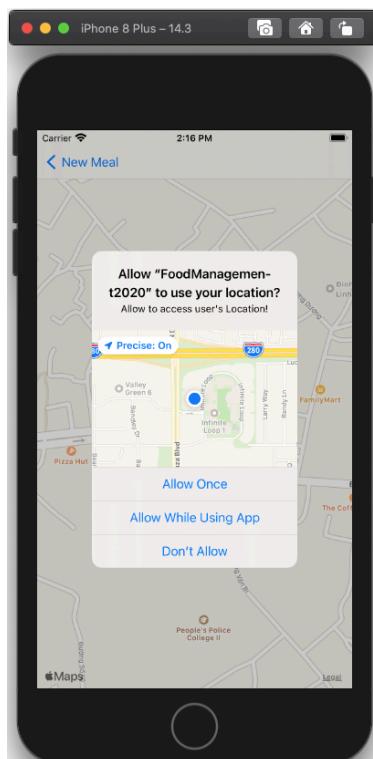
```
private var locationManager: CLLocationManager!
private var currentLocation: CLLocation?
```

Tiếp đến viết hàm lấy tọa độ hiện tại:

```
// Get current Location
func getCurrentLocation(){
    locationManager = CLLocationManager()
    locationManager.delegate = self
    // Define how to update location (immediate)
    locationManager.desiredAccuracy = kCLLocationAccuracyBest
    // Check for Location Services
    if CLLocationManager.locationServicesEnabled() {
        locationManager.requestWhenInUseAuthorization()
        locationManager.startUpdatingLocation()
    }
}
```

Hàm này sẽ được gọi trong hàm viewDidLoad. Tiếp đến viết hàm uỷ quyền thực hiện cập nhật vị trí hiện tại liên tục với hàm centerMapOnLocation là kết quả của bài tập 1:

```
//MARK: For Location
func locationManager(_ manager: CLLocationManager, didUpdateLocations
locations: [CLLocation]) {
    defer {
        currentLocation = locations.last
        locationManager.stopUpdatingLocation()
    }
    if currentLocation == nil {
        // Zoom to user location
        if let userLocation = locations.last {
            centerMapOnLocation(location: userLocation, radius: 500)
        }
    }
}
```



Chọn máy ảo => Chọn Features => Location => Custom Location => Nhập vào toạ độ hiện tại (Do máy ảo không có cảm biến). Thực hiện chạy chương trình, xuất hiện yêu cầu cho phép truy xuất vị trí hiện tại trên bản đồ (Hình 4.2.4.2), chọn cho phép (Allow While Using App).

Bài tập: Chạy chương trình trên máy ảo, chạy chương trình trên máy thật, di chuyển và cho nhận xét!

4.2.5 Tương tác với các Annotation trên bản đồ

Với các Annotation đã được đánh dấu trên bản đồ, ta có thể tùy biến chúng và người sử dụng có thể tương tác dễ dàng với các Annotation đó. Trước tiên cần thực hiện uỷ quyền các hàm trong MKMapViewDelegate.

Hình 4.2.4.2 Yêu cầu cấp quyền truy xuất

Bài tập: Tìm hiểu và thực hiện tương tác với các Annotation!

4.2.6 Xác định vị trí và di chuyển trên bản đồ

Bài tập: Dựa trên kết quả mục 4.2.4 chúng ta luôn lấy được vị trí hiện tại mà người dùng đang đứng trên bản đồ. Sửa chương trình ở mục 4.2.4 để cho mỗi khi người sử dụng di chuyển thì toạ độ của vị trí hiện tại cũng di chuyển theo trên bản đồ đó và ta có thể thấy được hướng di chuyển của chính người dùng trên bản đồ đó.

Gợi ý: Sử dụng lệnh hiển thị vị trí hiện tại của người dùng trên bản đồ:

```
// Show the current location as a blue point  
mapView.showsUserLocation = true
```

4.2.7 Đánh dấu vị trí bằng sự kiện LongPressGesture

Một trong những thao tác thường hay được định nghĩa cho các ứng dụng trên di động đó là bắt sự kiện **long press** (Tap và giữ tay một lúc trên màn hình điện thoại). Thực hiện như sau:

```
// For long press gesture  
func addLongPressGesture(){  
    let longPressRecognizer:UILongPressGestureRecognizer =  
        UILongPressGestureRecognizer(target: self,  
                                      action:#selector(handleLongPress(_)))  
    longPressRecognizer.minimumPressDuration = 1.0  
    mapView.addGestureRecognizer(longPressRecognizer)  
}
```

Trước tiên khai báo một đối tượng kiểu **UILongPressGestureRecognizer** trong đối tượng này sẽ truyền vào action là một selector trỏ đến hàm **handleLongPress** và mỗi khi xuất hiện sự kiện LongPress thì hàm đó sẽ được gọi. Hàm **handleLongPress** có tham số truyền vào là một **UIGestureRecognizer** cho phép nhận biết toạ độ của vị trí chõ ngón tay tap trên bản đồ. Đơn giản nhất ta sẽ viết hàm này sao cho mỗi khi người dùng LongPress tại một vị trí trên bản đồ thì sẽ đánh dấu một Annotation tại vị trí đó:

```
@objc func handleLongPress(_ gestureRecognizer:UIGestureRecognizer){  
    // Get the position of user's touch point  
    let touchPoint = gestureRecognizer.location(in: mapView)  
    let touchCoordinate = mapView.convert(touchPoint, toCoordinateFrom:  
                                            mapView)  
    markLocation = CLLocation(latitude: touchCoordinate.latitude,  
                               longitude: touchCoordinate.longitude)  
    let annotation = MyMapAnnotation(coordinate:  
                                       markLocation!.coordinate, title: "You mark here!", subtitle: "")  
    mapView.addAnnotation(annotation)  
}
```

4.2.8 Chỉ đường

Chỉ đường là một công việc tương đối phức tạp cho các ứng dụng trên di động. Trong MapKit cần thực hiện các bước sau:

Bước 1: Đánh dấu điểm đi và điểm đến trên bản đồ map

```
let sourcePlaceMark = MKPlacemark(coordinate: sourceCoordinate,  
                                     addressDictionary: nil)  
let destinationPlaceMark = MKPlacemark(coordinate:  
                                         destinationCoordinate, addressDictionary: nil)
```

Bước 2: Lấy MapItem cho điểm đi và điểm đến dựa trên kết quả bước 1.

```
let sourceMapItem = MKMapItem(placemark: sourcePlaceMark)
let destinationMapItem = MKMapItem(placemark: destinationPlaceMark)
```

Bước 3: Dựa trên các MapItem xác định các yêu cầu về hướng cho điểm đi và điểm đến.

```
let directRequest = MKDirections.Request()
directRequest.source = sourceMapItem
directRequest.destination = destinationMapItem
directRequest.transportType = .automobile
```

Bước 4: Tính toán các điểm theo hướng đã xác định ở bước 3 và tiến hành vẽ đường đi.

```
let direction = MKDirections(request: directRequest)
direction.calculate { (response, error) in
    if error == nil {
        if let route = response?.routes.first {
            self.mapView.addOverlay(route.polyline, level: .aboveRoads)
            let rect = route.polyline.boundingMapRect
            self.mapView.setVisibleMapRect(rect, edgePadding:
                UIEdgeInsets.init(top: 40, left: 40, bottom: 20,
                                  right: 20), animated: true)
        }
    } else {
        print(error?.localizedDescription ?? "Unknown error!")
    }
}
```

Bước 5: Định nghĩa lại hàm vẽ nối điểm trên bản đồ với nét vẽ màu xanh dương, độ rộng nét vẽ 3:

```
// Draw the route
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) ->
MKOverlayRenderer {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.strokeColor = UIColor.blue
    renderer.lineWidth = 3.0
    return renderer
}
```

Điều chỉnh lại hàm **handleLongPress** để mỗi khi sự kiện LongPress xuất hiện sẽ lấy toạ độ trên bản đồ của điểm được tap, thêm vào Annotation cho nó và thực hiện vẽ đường đi trong hàm chỉ đường 4 bước ở trên:

```
// Get the position of user's touch point
let touchPoint = gestureRecognizer.location(in: mapView)
let touchCoordinate = mapView.convert(touchPoint, toCoordinateFrom:
mapView)
markLocation = CLLocation(latitude: touchCoordinate.latitude,
longitude: touchCoordinate.longitude)
// Clear all previous Annotations
mapView.removeAnnotations(mapView.annotations)
if let meal = self.meal {
```

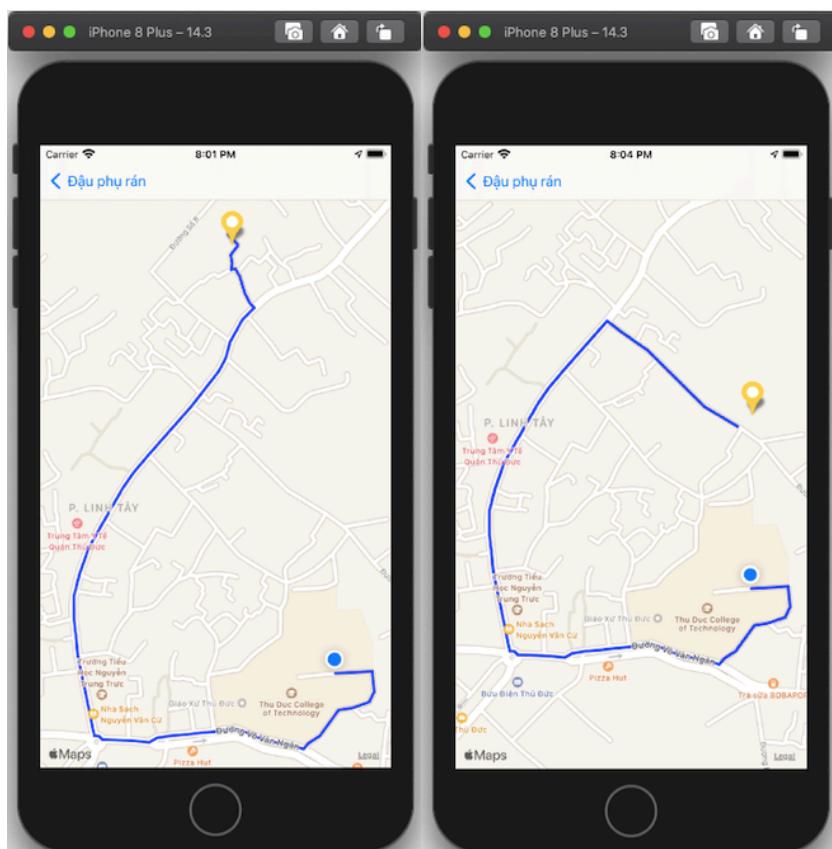
```

        addAnnotation(coordinate: touchCoordinate, title: meal.mealName,
                      subtitle: "", type: MapAnnotationType.specific)
    }
else {
    addAnnotation(coordinate: touchCoordinate, title: defaultMess,
                  subtitle: "", type: MapAnnotationType.specific)
}
mapView.removeOverlays(mapView.overlays)
routeFromSourceToDes(sourceCoordinate: (currentLocation?.coordinate)!,  

                      destinationCoordinate: touchCoordinate)

```

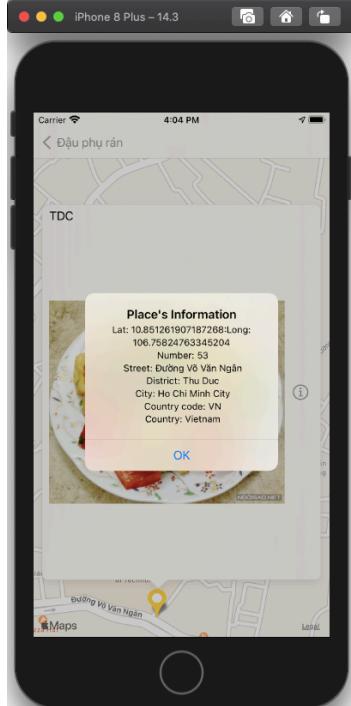
Kết quả khi chạy chương trình (Hình 4.2.8.1):



Hình 4.2.8.1 Màn hình MapView với chức năng chỉ đường

4.3 Câu hỏi và bài tập chương 4

1. Trong mục 4.2.2 sau khi khởi tạo và hiển thị bản đồ trên màn hình điện thoại, hãy viết hàm thực hiện công việc sau: Truyền vào hàm hai tham số tọa độ của điểm cần đến (Có kiểu là **CLLocation**) và phạm vi khu vực cần hiển thị (Có kiểu là **CLLocationDistance**),



khi gọi hàm bản đồ sẽ di chuyển đến đúng tọa độ được truyền bởi tham số thứ nhất và bao phủ một vùng xung quanh tọa độ đó với khoảng cách bao phủ bằng tham số thứ 2.

2. Cải tiến chương trình trong mục 4.2.3 để đánh dấu các danh lam thắng cảnh nổi tiếng của Việt Nam và trên thế giới!

3. Thực hiện bài tập trong 4.2.5 sao cho mỗi khi chạy ứng dụng, tap vào nút “Go to Map” sẽ chuyển sang màn hình MapViewController, đánh dấu tại một điểm xác định (vị trí của quán có bán món ăn đó). Mỗi khi tap lên điểm Marker đó sẽ hiển thị ảnh của món ăn tương ứng, nếu tap lên nút (!) sẽ hiển thị thông tin về quán (Hình 4.3.1).

Hình 4.3.1 Cải tiến chức năng 4.2.5

4. Cải tiến chương trình trong mục 4.2.7 sao cho tại vị trí hiện tại người dùng đang đứng trên bản đồ sẽ đánh dấu bằng một Annotation (khác màu sắc, hình ảnh...) và mỗi khi người sử dụng LongPress tại một vị trí nào khác trên bản đồ thì sẽ được đánh dấu bằng một Annotation khác.

5*. Sửa chương trình để mỗi khi từ màn hình MealDetail chuyển sang màn hình MapView thì thông tin về món ăn sẽ được truyền sang màn hình MapView. Nếu món ăn đó đã có tọa độ của quán bán nó thì sẽ hiển thị một Annotation tại vị trí của quán ăn trên bản đồ. Nếu tương tác vào Annotation này sẽ biết các thông tin khác về quán ăn đó. Nếu đó là món ăn mới đang thêm vào danh sách và chưa có tọa độ của quán trên bản đồ, thì mỗi khi LongPress lên một vị trí nào đó sẽ lấy tọa độ điểm đó làm tọa độ của quán ăn bán nó.

6*. Cải tiến chương trình mục 4.2.8 và kết hợp với chương trình cải tiến trong bài tập số 5 ở trên điều chỉnh chương trình sao cho nếu đã có tọa độ quán bán món ăn thì sẽ vẽ đường đi từ vị trí hiện tại đến quán ăn đó, còn nếu Long Press tại một vị trí khác sẽ hiển

thì Dialog hỏi xem người dùng có muốn thay đổi địa chỉ quán hay không? Nếu muốn sẽ cập nhật địa chỉ mới. Nếu món ăn chưa có tọa độ, mỗi khi LongPess sẽ hỏi xem có muốn lấy tọa độ làm địa chỉ quán hay không? Nếu muốn, sẽ cập nhật địa chỉ mới.

TÀI LIỆU THAM KHẢO

- [1] Neil Smyth, **iOS 11 App Development Essentials**, eBookFrenzy, 2018
- [2] Vandan Nahavandipoor, **iOS 10 Swift Programming Cookbook**, O'Reilly Media USA, 2017
- [3] Stefan Kaczmarek, Brad Lees, Gary Bennett, **Swift 5 for Absolute Beginners: Learn to Develop Apps for iOS**, Apress 2019