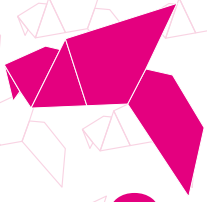
飛岡 辰哉 著



Swift 4

プログラミング入門

iOS 11＋Xcode 9 対応

読者特典
ボーナス
PDF
非売品

ソシム

本PDFについて

- **本 PDFは書籍『Swift 4プログラミング入門 iOS 11＋ Xcode 9 対応』(ISBN978-4-8026-1153-4)をご購入いただいたお客様のために「ボーナスPDF」として作成されています。**
- **本 PDFの最新版は上記書籍のサポートサイトより入手できます。本 PDFの内容は予告なく更新されることがあります。最新の情報につきましては、サポートサイトをご覧ください。**
- **本 PDFは、内容と形態を変更しない限り、自由に配布してくださって構いませんが、上記書籍のご購入を前提とした説明が多く含まれます。予めご了承ください。**
- **本PDF中の表現で「書籍版」とあるのは、上記書籍を指します。**
- **本PDFの内容は無保証です。本PDFを販売することを禁じます。**
- **本PDFを印刷することはできません。**
- **PDF中のURLはハイパーリンクとして機能します。**
- **PDF中の各項目には「しおり」が設定されています。目次の代わりにお使いください。**

サンプルプログラムについて

サンプルプログラムは、以下からダウンロードできます。

https://github.com/tnantoka/swiftbook-examples/archive/1.0.zip

MITライセンスです。自己責任になりますが自由にご利用くださって構いません。また、配布しているサンプルコード内にある「/// [marker数字]」というコメントは執筆上のマーキングであり、プログラム上の意味はありません。

一部のサンプルでは、以下のフォントを使用しています。

- **http://fontawesome.io/（SIL Open Font License）**
- **https://fonts.google.com/specimen/Italiana（SIL Open Font License）**

本PDFでサンプルプログラムを示す場合は、以下のようにサンプルプログラムの場所を示します。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter1/Icon.xcodeproj
Storyboard	Bonus/Chapter1/Icon/Icon/Base.lproj/LaunchScreen.storyboard
ソース	変更しません

『Swift 4プログラミング入門』ボーナスPDF

Chapter 1

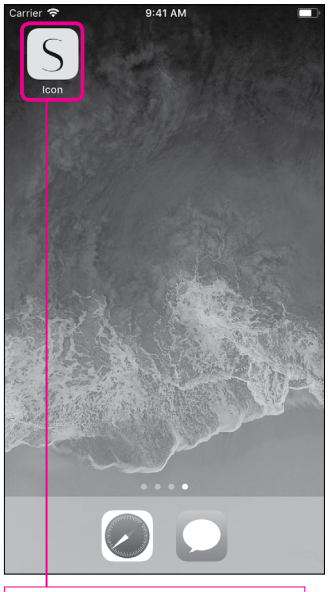
App Storeにリリースする

本 PDFの Chpater 1では作成したアプリを App Storeにリリースするための手順を紹介します。

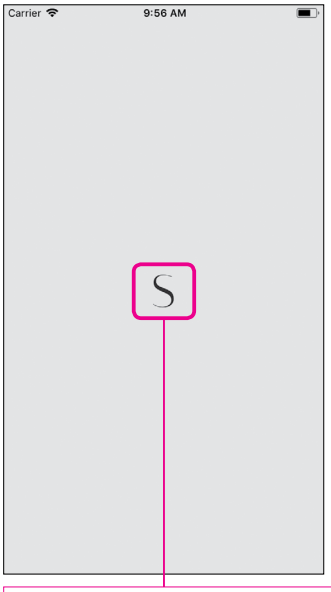
Section 1.1

アイコン・起動画面の準備

サンプルアプリ



アプリにアイコンを設定します。



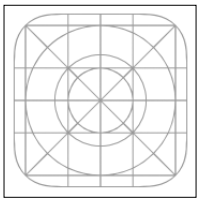
アプリの起動時に独自のLaunchScreenを表示させます。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter1/Icon.xcodeproj
Storyboard	Bonus/Chapter1/Icon/Icon/Base.lproj/LaunchScreen.storyboard
ソース	変更しません

iPhoneアプリに必要なアイコン

アプリをインストールすると、デフォルトでは以下のようなアイコンがホーム画面に表示されます。また、起動画面については真っ白です。独自のものを設定しましょう

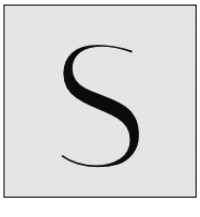


iPhoneアプリには次の表にあるサイズのアイコンを設定する必要があります。

サイズ	必須
40px	−
58px	−
60px	−
80px	−
87px	−
120px	○
180px	○
1024px	○（Xcode9から）

必須に○がついているものは、設定しないと Appleへ提出する際エラーになります。他のものもできるだけ設定しておいた方がいいでしょう。

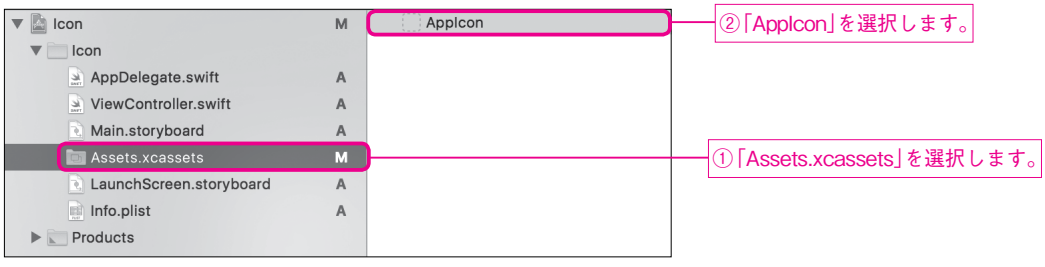
今回は以下のようなアイコンを必須の３サイズ（120px、180px、1024px）用意しました。

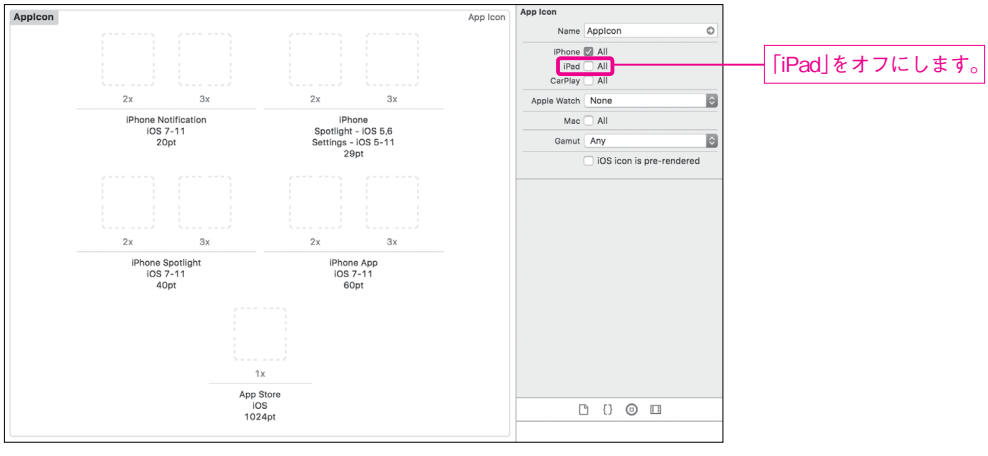


Note

残念ながらXcodeにアイコンの作成機能は備わっていないので、画像編集ツールで作成しましょう。
筆者は、書籍版の Chapter 15で紹介した図形描画を応用し、本 PDFの Chapter 3で紹介する Xcode Playgroundでアイコンを作成しました。
https://github.com/tnantoka/IconCreator

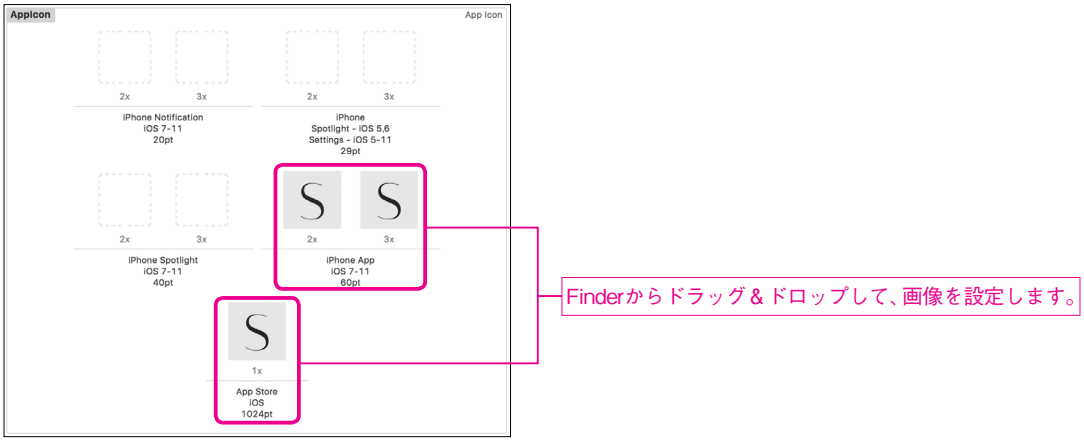
アイコンの設定は`Assets.xcassets`の`AppIcon`から行います。





それぞれのサイズのところにドラッグ＆ドロップして設定します。サイズが少しわかりづらいですが、それぞれ以下のサイズのアイコンを指定します。

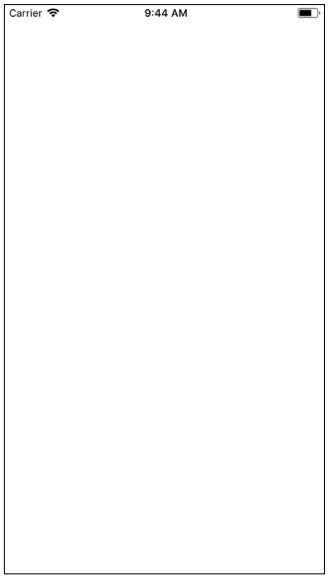
- **「60pt」の「2x」：「60×2＝120px」**
- **「60pt」の「3x」：「60×3＝180px」**
- **「1024pt」の「1x」：「1024×1＝1024px」**

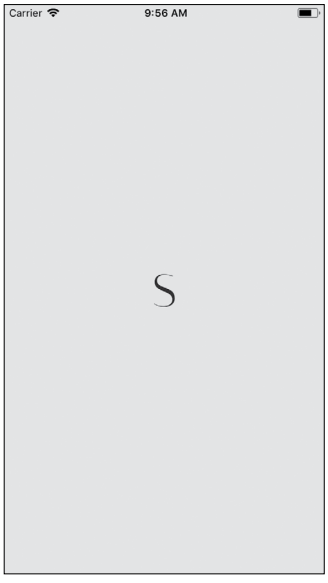


この状態で実行すると、ホーム画面のアイコンが独自のものに変わります。

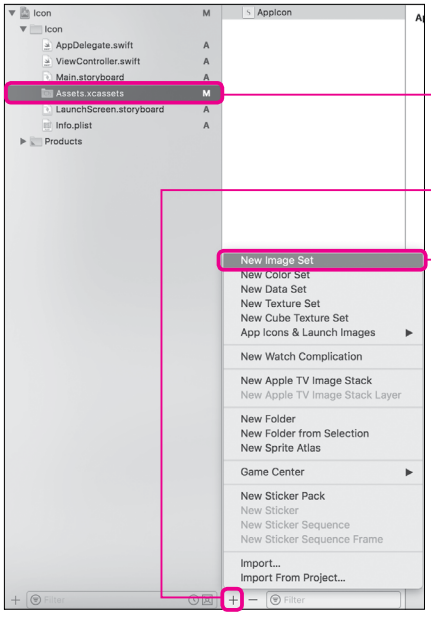
起動画面

次は起動画面を変更します。デフォルトでは以下のように真っ白になっています。これを中央にアイコンを置いたシンプルなものに変更します。





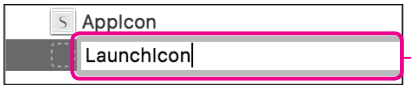
「Assets.xcassets」に新規画像セットを追加します。名前は「LaunchIcon」に変更します。



①「Navigator Area」の「Project Navigator」で「Assets.xcassets」を選択します。

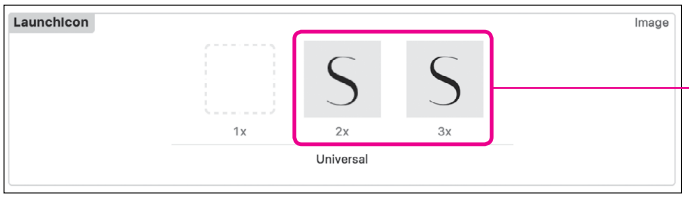
②「＋」をクリックします。

③「New Image Set」を選択します。



空の画像セットをダブルクリックして
名前を「LaunchIcon」に変更します。

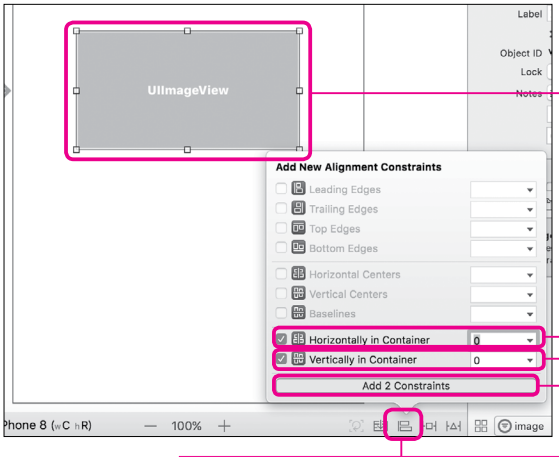
「2x」「3x」に、それぞれ「120px」「180px」のアイコン画像を設定します。「iPhoneアプリに必要なアイコン」で使った画像と同じです。



ドラッグ＆ドロップで画像を設定します。

設定は「LaunchScreen.storyboard」で行います。操作方法は普通のStoryboardと一緒です。

Image Viewを中央に配置します。どのサイズのデバイスでも中央に表示されるように AutoLayoutで制約を設定します。



①Image Viewをドラッグし、ガイドに合わせて
垂直方向、水平方向中央に配置します。

③「Horizontally in Container」をオンにし、
値「0」を設定します。

④「Vertically in Container」をオンにし、
値「0」を設定します。

⑤「Add 2 Constraints」ボタンを押します。

②Image Viewを選択した状態で「Align」を選択します。

Image Viewに画像を設定します。Utility Areaの「Image」に先ほど追加した「LaunchIcon」を設定します。

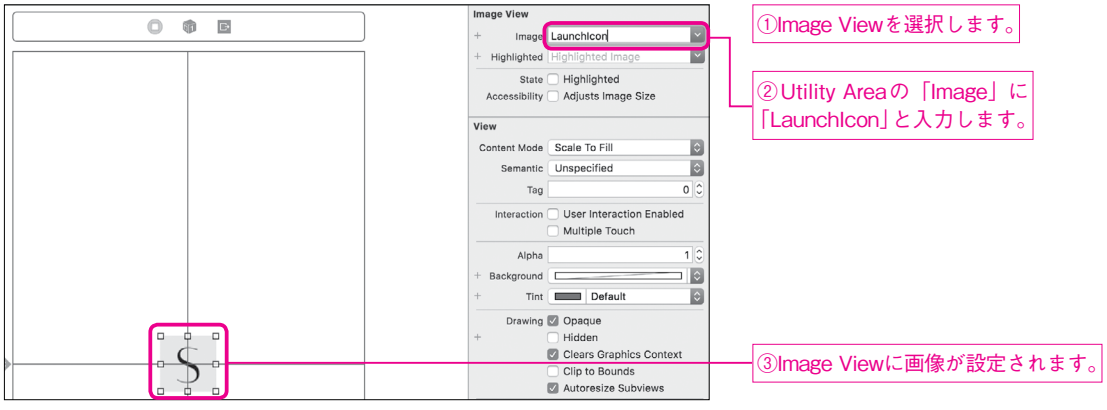
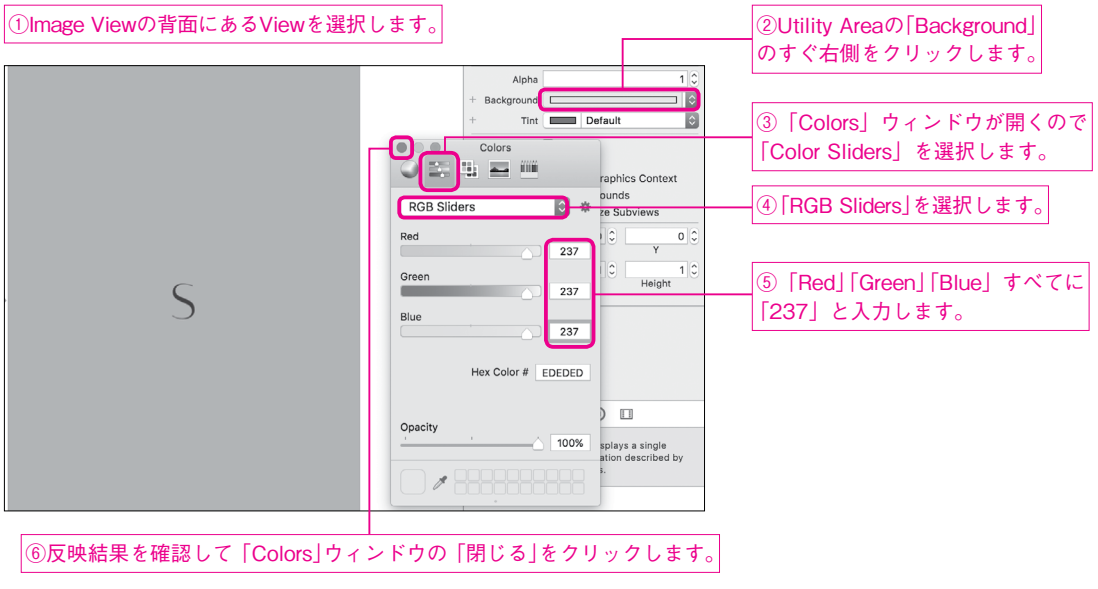


Image Viewの背面にあるViewの背景色をアイコンの背景色と揃えます。アイコンの背景色はRGB（237，237，237）で制作してあります。



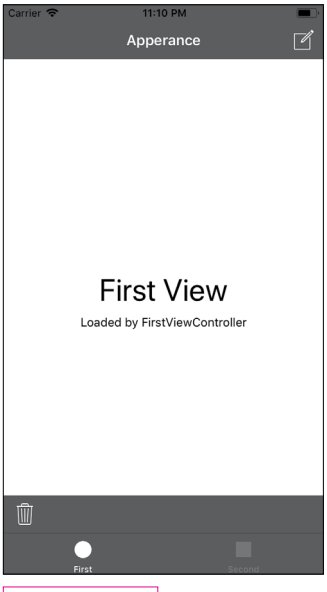
これでアプリの起動時に独自のLaunchScreenが表示されるようになります。

Section 1.2

デザインを整える

サンプルアプリ

デザイン変更前



デザイン変更後

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter1/Appearance.xcodeproj
Storyboard	Bonus/Chapter1/Appearance/Appearance/Base.lproj/Main.storyboard
ソース	Bonus/Chapter1/Appearance/Appearance/AppDelegate.swift Bonus/Chapter1/Appearance/Appearance/FirstViewController.swift Bonus/Chapter1/Appearance/Appearance/NavigationController.swift

デザイン変更の基礎知識

開発メンバーにデザイナーがいる場合は、ぜひデザインを依頼し、そのデザインに合わせて開発を進めましょう。

デザイナー以外が下手に手を加えると、何も手を加えない状態よりも使いにくくなってしまう可能性もありますから、注意しましょう。開発メンバーにデザイナーがいない場合は、色をアイコンに合わせる程度の調整で留めておくのもよいでしょう。

ここでは各種バーの色を変える手順を紹介します。

Note

デザインを変更する際は、Apple社が定めたガイドラインから外れないように気をつけてください。
『Human Interface Guidelines』
https://developer.apple.com/ios/human-interface-guidelines/overview/themes/

UIAppearance

`UIAppearance`は各UI部品のデフォルト設定を変えるときに使うクラスです。

各UI部品が表示される前に設定を変更する必要があるので、`AppDelegate`の`application(_:didFinishLaunchingWithOptions:)`内に設定を変更する処理を書きます。次のコメントが目印です。

```
// Override point for customization after application launch.
```

`UIApperance`の他にも、データベースの初期化などアプリ起動時に行いたい処理はこの場所に記述することが多いです。

`UIAppearance`の書式は次の通りです。

書式　対応バージョン v3 v4

```
クラス名.appearance().変更するプロパティ = 設定値
```

以降では`UIAppearance`の機能を使ってバーの色を変更します。

デフォルトデザインの作成

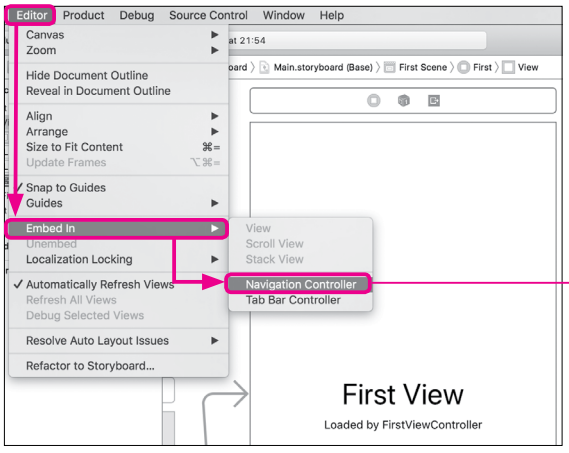
3つのバーがあるアプリを作成します。

- **タブバー（Tab Bar）**
- **ナビゲーションバー（Navigation Bar）**
- **ツールバー（Toolbar）**

設定を簡単にするため、ここでは「Tabbed App」テンプレートを使います。

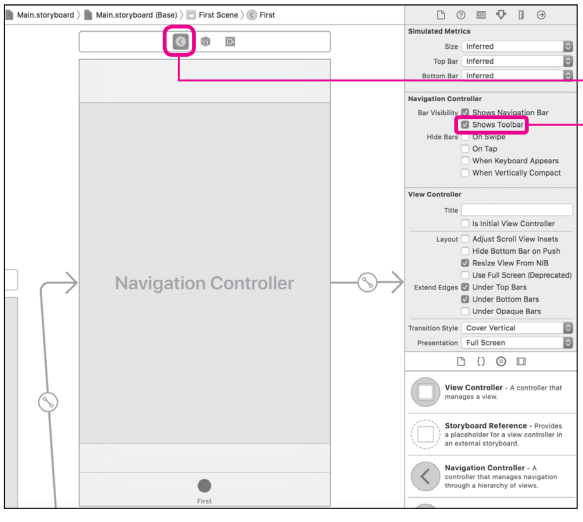
これだけで、タブバーを表示させることができました。

次に、ナビゲーションバーを表示させるために、「First」ビューコントローラーを「Navigation Controller」へ埋め込みます。「Navigator Area」の「Project Navigator」で「Main.storyboard」を選択し、「Document Outline」で「First View Controller Scene」の左側の三角形をクリックします。表示された「First」ビューコントローラーを選択し、メニューバーから「Editor」→「Embed In」→「Navigation Controller」を選択します。



メニューバーから「Editor」→「Embed in」→「Navigation Controller」の順で選択します。

さらに、ツールバー（Toolbar）を表示させます。「Main.storyboard」で「Navigation Controller」を選択した状態で、「Utility Area」の「Shows Toolbar」をオンにします。

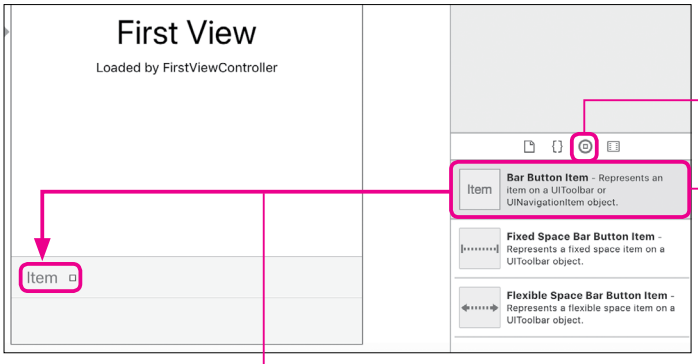


①「Navigation Controller」を選択します。

②「Shows Toolbar」をオンにします。

バーだけですと味気ないので、ナビゲーションバーとツールバーにアイテム（Bar Button Item）を追加します。ここでは見た目のカスタマイズをしたいだけですので、特にアクションなどは設定しません。

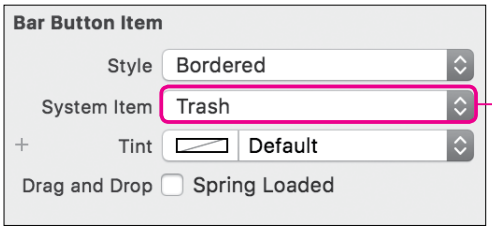
ツールバーに「ゴミ箱（Trash）」ボタンを追加してみます。



①「Show the Object library」をクリックします。

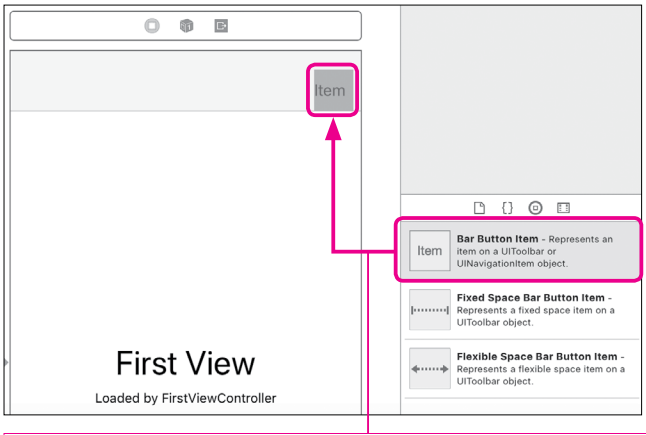
②「Bar Button Item」(barb と検索窓に入力すると見つかります) を探します。

③「Bar Button Item」を「Main.storyboard」の「First View」に表示されているツールバーへドラッグ & ドロップします。

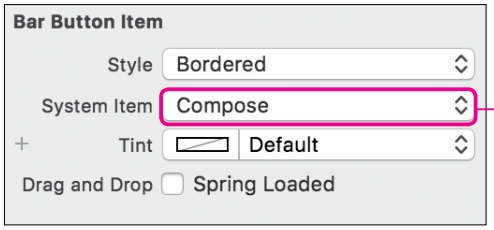


「Bar Button Item」を選択した状態 (Itemの右側に□が表示されている状態) で、「Utility Area」の「System Item」を「Trash」に変更します。

これで、ツールバーに「ゴミ箱」ボタンを追加できました。
続いてナビゲーションバーに、「作成 (Compose)」ボタンを追加してみます。



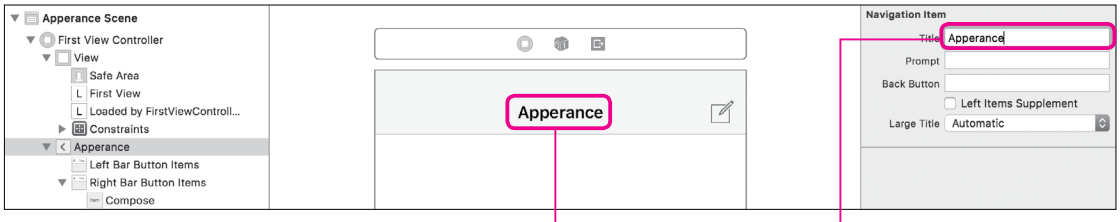
「Bar Button Item」を「Main.storyboard」の「First View」に表示されているナビゲーションバーへドラッグ & ドロップします。



「Bar Button Item」を選択した状態（Itemの右側に□が表示されている状態）で、「Utility Area」の「System Item」を「Compose」に変更します。

これで、ナビゲーションバーに「作成」ボタンを追加できました。

最後に、ナビゲーションアイテム（Navigation Item）にタイトルをつけておきます。



①ナビゲーションバーの中央付近をクリックして選択します。

②「Utility Area」の「Title」に「Appearance」と入力します。

以上で準備は終わりです。何も設定しないと３つのバーの背景色はすべて薄いグレーで表示されています。

ナビゲーションバー

ツールバー

タブバー

NavigationBarのデザイン変更

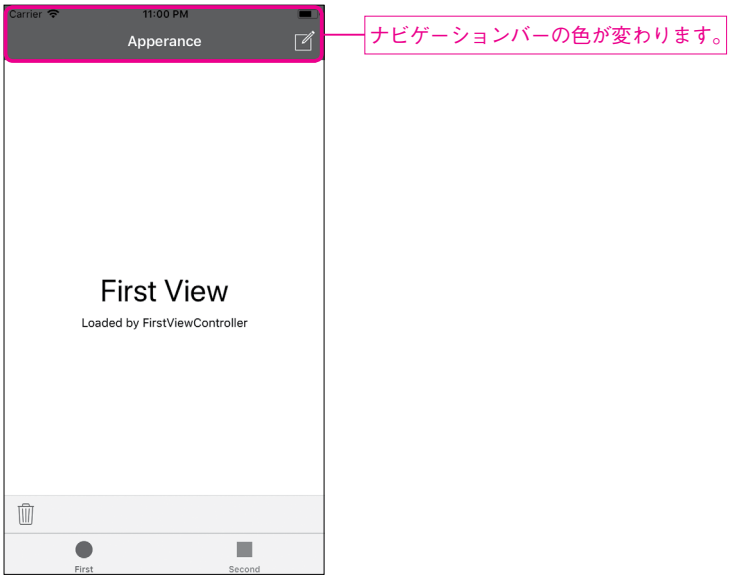
ナビゲーションバーのデザイン変更から始めます。「Navigator Area」の「Project Navigator」で「AppDelegate.swift」を選択し、Section 21.2の先頭で目印としたコメントを探し、次のプログラムを入力します。

サンプルプログラム　Bonus/Chapter1/Appearance/Appearance/AppDelegate.swift

```
// Override point for customization after application launch.
UINavigationBar.appearance().barTintColor = .darkGray
UINavigationBar.appearance().tintColor = .white
UINavigationBar.appearance().titleTextAttributes = [
    NSAttributedStringKey.foregroundColor: UIColor.white
]
```

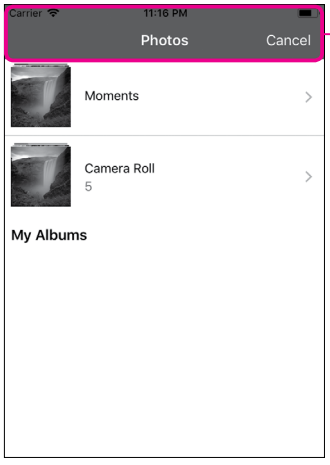
`barTintColor`が背景色、`tintColor`が左右のボタンの色、`titleTextAttributes`は中央のタイトルの設定です。

このように設定すると、ナビゲーションバーの色が変わります。



注意

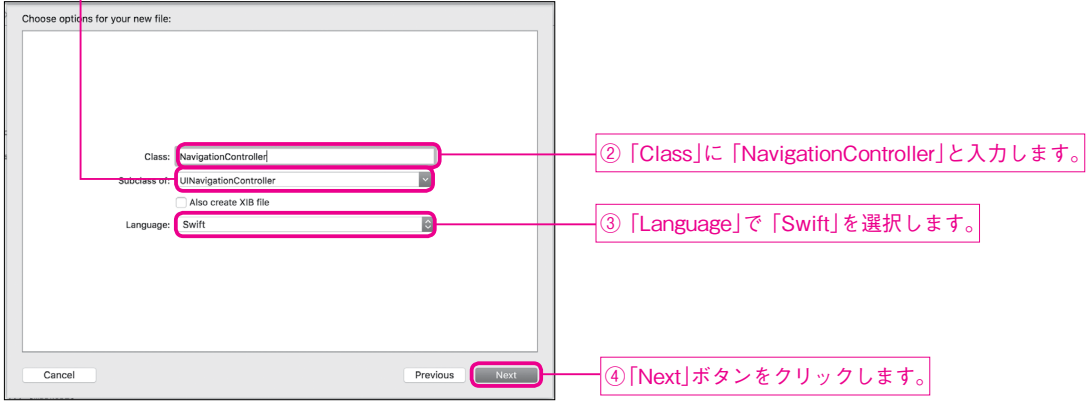
`UINavigationBar`全体のデザインを変えると、予期せぬ部分に影響を与えてしまう可能性があります。例えば、`UIImagePickerController`のバーも色が変わってしまいます。



意図しない場所の色も変わってしまいました。

`UINavigationController`を継承した `NavigationController`を使えば、自分の作成したバーだけにAppearanceを設定することができます。

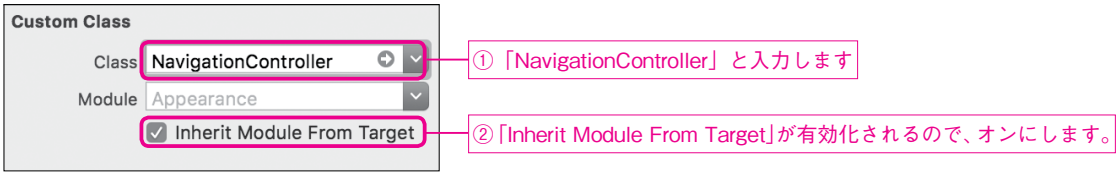
メニューバーから「File」→「New」→「File」と選択し、新規ファイルのテンプレート選択画面を表示させます。表示された画面で「Cocoa Touch Class」テンプレートを選択し、「Next」ボタンを押します。



保存先を指定するダイアログが表示されるので、プロジェクト名と同じ名前のフォルダーを指定して「Create」ボタンをクリックすれば、`NavigationController.swift`を作成できます。

「Navigator Area」の「Project Navigator」で「Main.storyboard」を選択し、「Navigation Controller」を選択します。

「Utility Area」上部に並ぶボタンのうち左から3番目の「Show the identity inspector」ボタンをクリックし、カスタムクラスを指定します。



カスタムクラス配下のナビゲーションバーのみにデザインの変更が反映されるように Appearanceの書き方を変更します。書式は以下のとおりです。

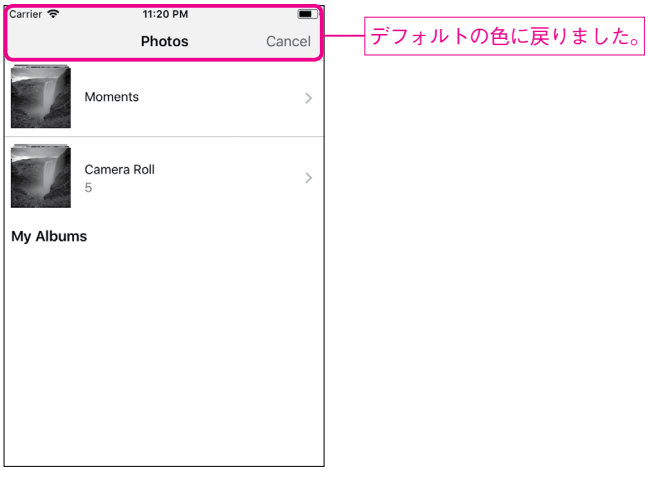
書式 対応バージョン v3 v4

```
クラス名.appearance(whenContainedInInstancesOf: [対象のクラス.self]).変更するプロパティ = 設定値
```

サンプルプログラム Bonus/Chapter1/Appearance/Appearance/AppDelegate.swift

```
UINavigationBar.appearance(whenContainedInInstancesOf:
    [NavigationController.self]).barTintColor = .darkGray
UINavigationBar.appearance(whenContainedInInstancesOf:
    [NavigationController.self]).tintColor = .white
UINavigationBar.appearance(whenContainedInInstancesOf:
    [NavigationController.self]).titleTextAttributes = [
    NSAttributedStringKey.foregroundColor: UIColor.white
]
```

これで、UIImagePickerControllerはデフォルト色のままになりました。



ToolBarのデザイン変更

ツールバーもナビゲーションバーと同様にデザインを変更できます。

サンプルプログラム　Bonus/Chapter1/Appearance/Appearance/AppDelegate.swift

```
UIToolbar.appearance().barTintColor = .darkGray
UIToolbar.appearance().tintColor = .white
```

TabBarのデザイン変更

タブバーもナビゲーションバーと同様にデザインを変更できます。

サンプルプログラム　Bonus/Chapter1/Appearance/Appearance/AppDelegate.swift

```
UITabBar.appearance().barTintColor = .darkGray
UITabBar.appearance().tintColor = .white
```

このようにバーの色を変更するだけでもアプリのイメージが変わります。

Note

バーの色を変えたあとはボタンの色との相性を必ず確認しましょう。バーの色だけを変えるとボタンが見づらくなってしまう場合があります。

Section 1.3

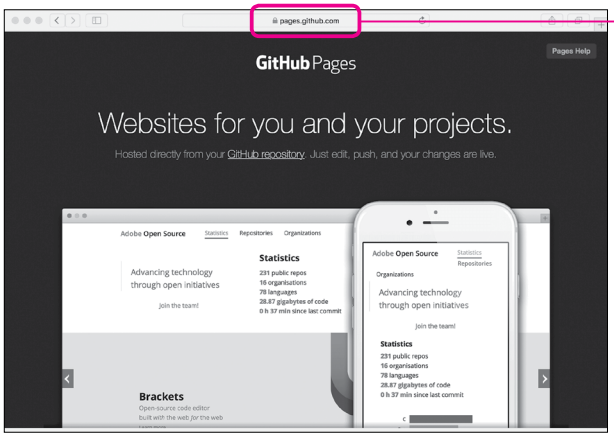
サポートサイトの開設

アプリを申請するためには、ユーザーが連絡の取れるサポートサイトが必要です。

企業であればコーポレートサイトを指定しておけば問題ありませんが、アプリ専用のランディングページなどを用意する方が望ましいでしょう。Twitterアカウントや Facebookページを指定しても大丈夫なので、手間をかけたくない場合はそれでも構いません。

GiHub Pages

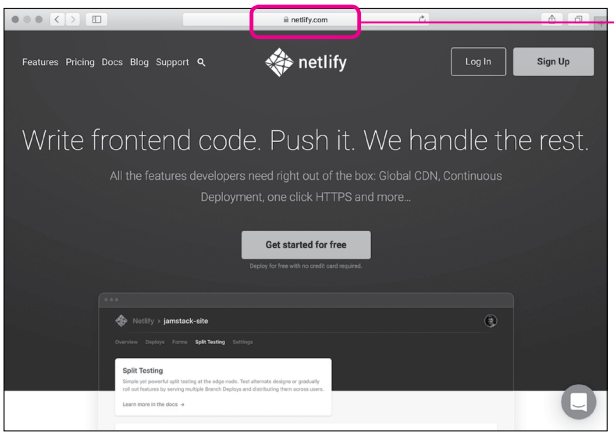
GitHub Pagesを使えば無料でサポートサイトをホスティングしてもらえます。日常的にホスティングサービスなどを利用していない場合には検討してみてください。



「GitHub Pages」
https://pages.github.com/

Netlify

筆者は無料で独自ドメインのHTTPS化もできる**Netlify**を利用しています。

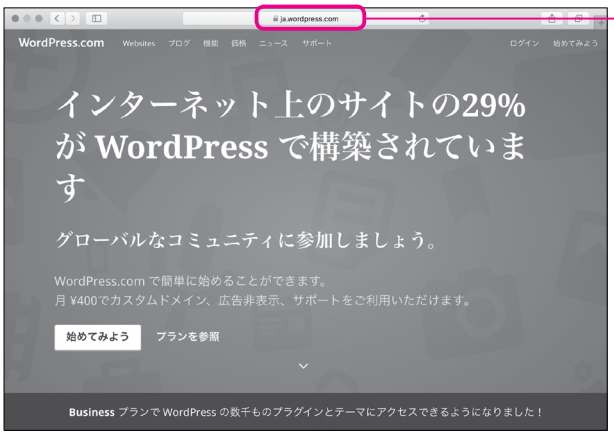


「Netlify」
https://www.netlify.com/

ブログサービス

WordPress.comなどのブログや CMS（コンテンツマネージメントシステム）のホスティングサービスを使う選択肢もあります。

このタイプのサービスは広告が表示される代わりに無料で利用できるというものが多いです。



「WordPress.com」
https://ja.wordpress.com/

Section 1.4

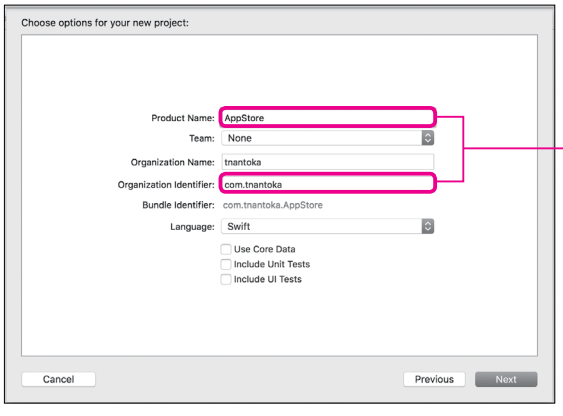
審査に提出する

App IDとBundle IDの取得

アプリには App ID というものを割り当てる必要があります。App IDは Apple Developerのサイトにログインして作成します。

その際にが、アプリを一意に識別する Bundle ID という情報が使われます。

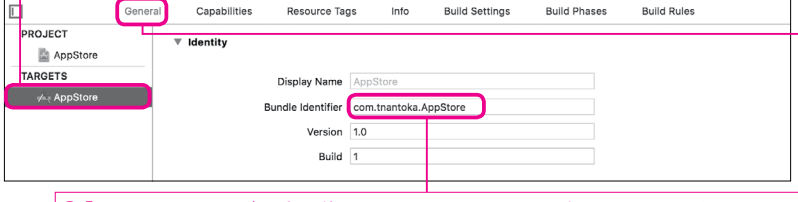
Bundle IDはプロジェクト作成時に入力した情報から自動的に作成されますが、後から変更することもできます。



このようにプロジェクトを作成した場合、「com.tnantoka.AppStore」が Bundle IDとなります。

プロジェクト作成後に Bundle IDを変更するには、「Navigator Area」の「Project Navigator」で、一番上の「プロジェクト」を選択します。

①「TARGETS」でプロジェクト名と同じターゲットを選択します。



②「General」を選択します。

③「Bundle Identifier」を書き換えることでBundle IDを変更することができます。

設定したBundle IDがわからなくなった場合には、この画面を確認するとよいでしょう。

審査提出の基本的な流れ

審査提出は次のような流れで進めます。Apple社のアプリ申請 Webサイトのデザインは変更されることがありますが、申請提出の流れはほとんど変わっていません。

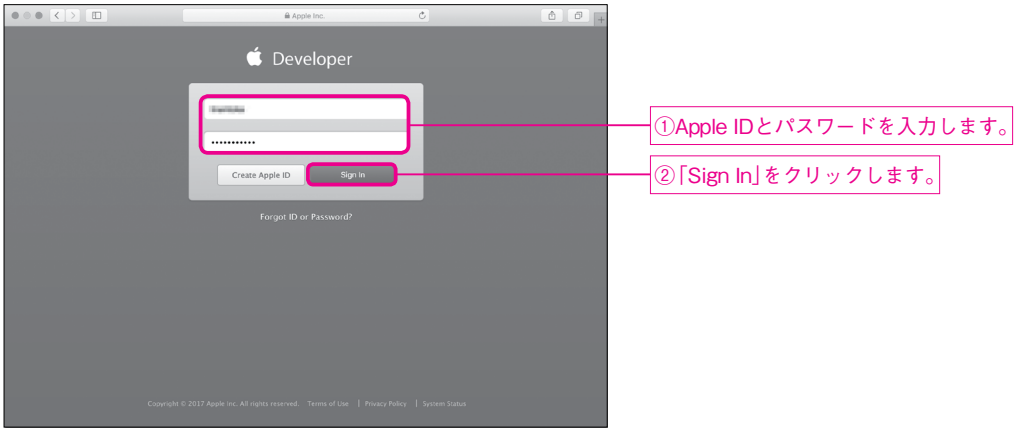
1. **App ID作成**
2. **チーム設定**
3. **バージョン作成**
4. **Archiveして送信**
5. **スクリーンショット**
6. **提出**

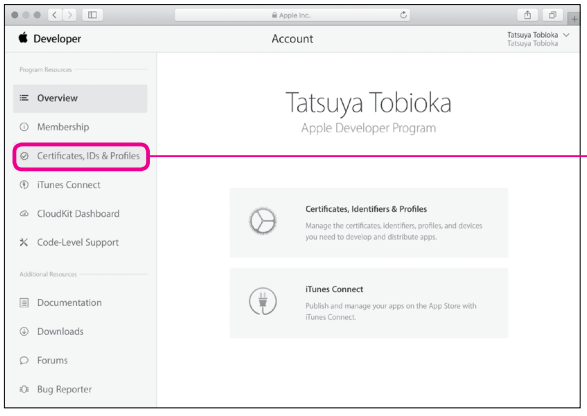
Note

申請にあたっては、最新バージョンの Safariを使うのが無難です。Chromeや Firefoxでは正しく動かないことがあるためです。

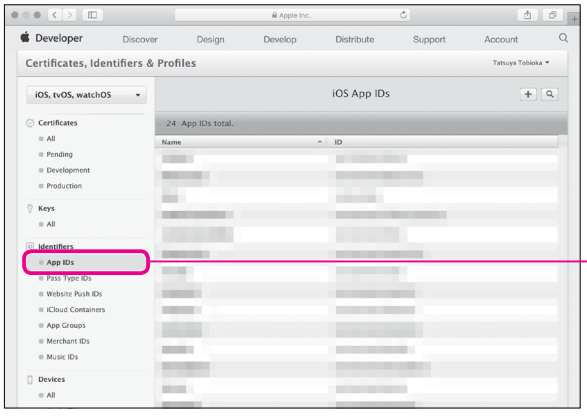
1. App ID作成

Apple開発者用のサイト（https://developer.apple.com/account/）でApp IDを作成します。

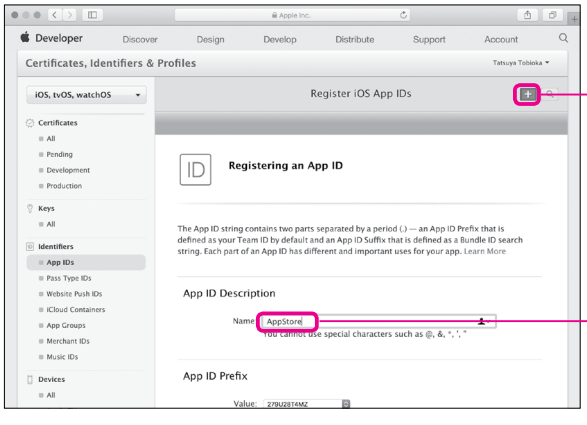




「Certificates, IDs & Profiles」を選択します。

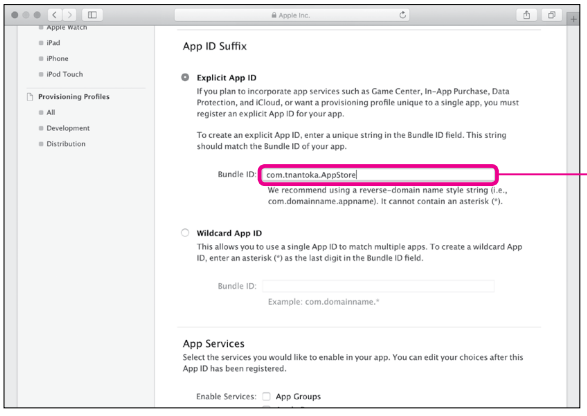


「App Ids」を選択します。

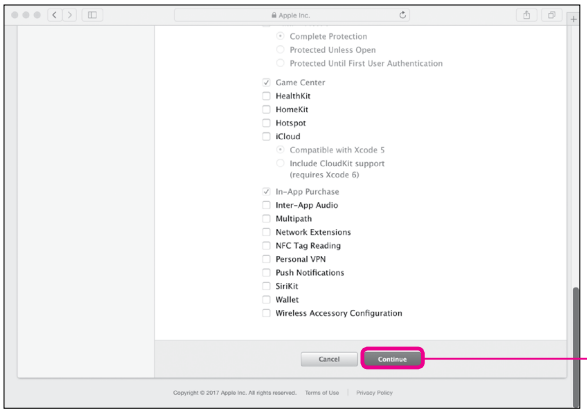


①「＋」をクリックして新規作成します。

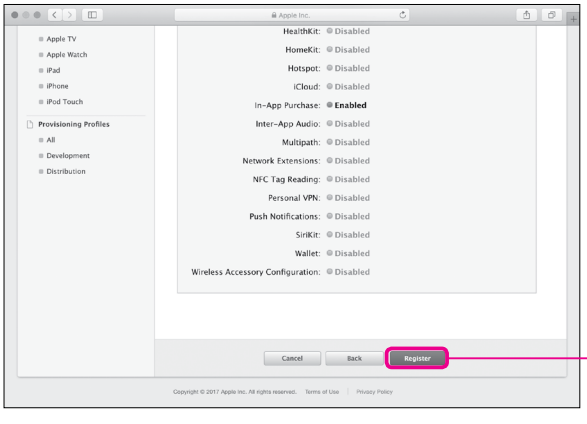
②「Name」を入力します。



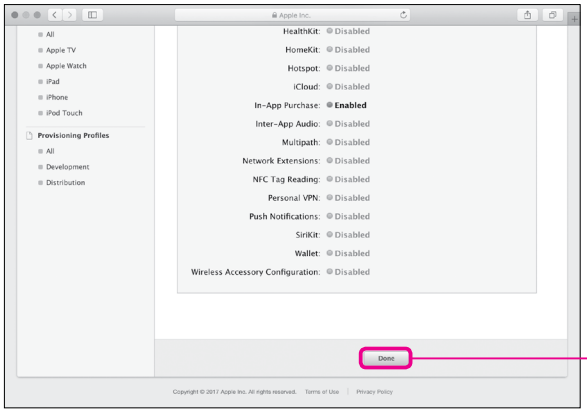
「Bundle ID」を入力します。



「Continue」をクリックします。



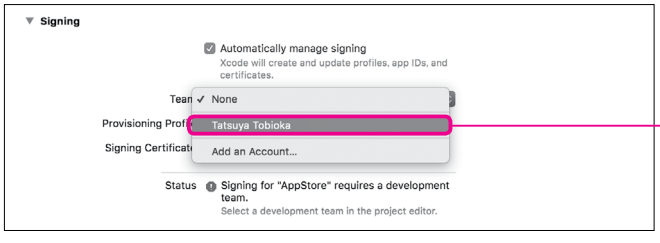
「Register」をクリックします。



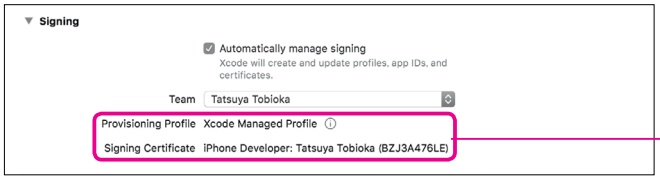
「Done」と表示されればApp IDの作成は完了です。

2. チーム設定

App Storeで配布するアプリは、証明書を使って署名をする必要があります。難しそうに聞こえますがXcodeがほぼ自動でやってくれます。その準備として書籍版の Chapter 19の実機検証と同様に Teamを選択しておきます。



「Team」を選択します。

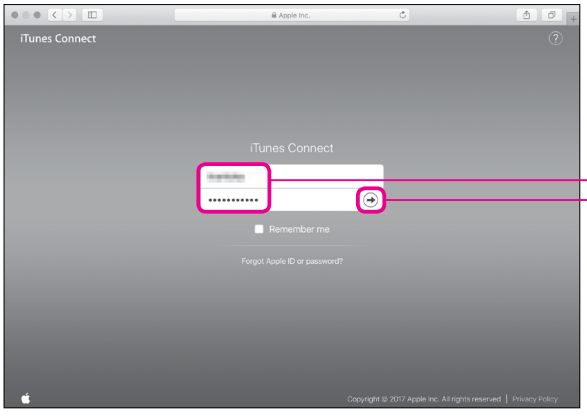


「Provisioning Profile」「Signing Certificate」が設定されます。

3. バージョン作成

アプリをアップロードするためには、**iTunes Connect**上でアプリのバージョンを用意しておく必要があります。

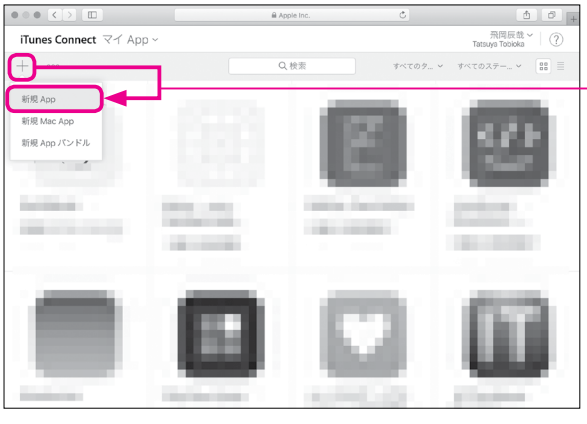
iTunes Connect（https://itunesconnect.apple.com/login）にログインします。



①Apple IDとパスワードを入力します。

②「→」をクリックします。

「マイ App」を選択します。

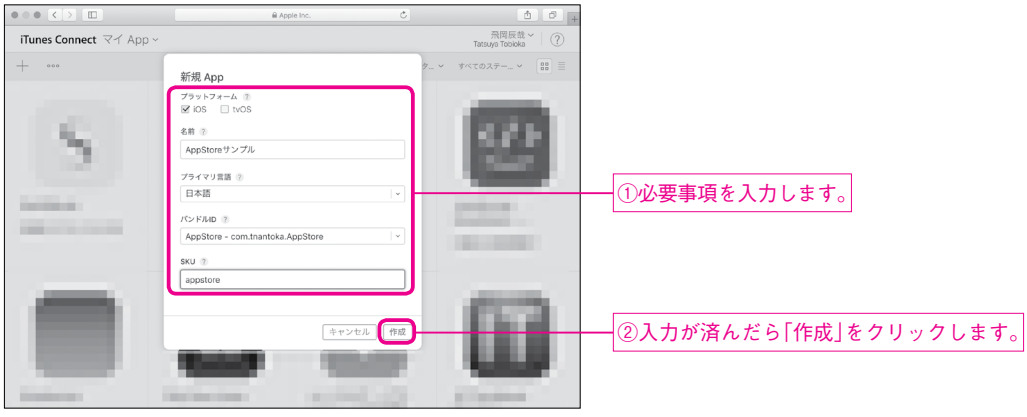


画面左上の「＋」をクリックして表示される
メニューから「新規 App」を選びます。

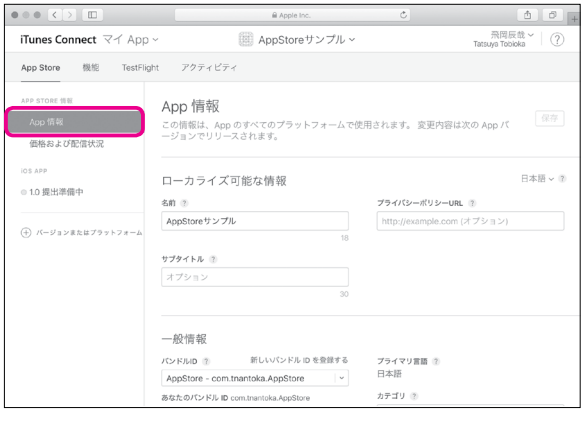
必要事項を入力します。

「名前」は AppStoreでの表示名で、すでに登録されるアプリの名前と重複した名前を付けることはできません。ここでは「AppStore」がすでに使われていたため「AppStoreサンプル」という名前で登録することにしました。

「バンドルID」では、先ほど作成したApp IDを選択します。



「1.0 提出準備中」と表示されれば、バージョンの作成は完了です。



Note

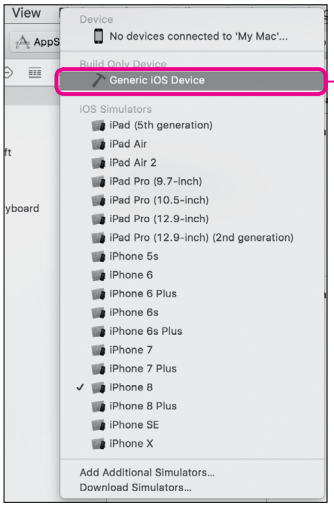
バージョンアップの場合は左下の「+バージョンまたはプラットフォーム」からバージョンを追加します。

4. Archiveして送信

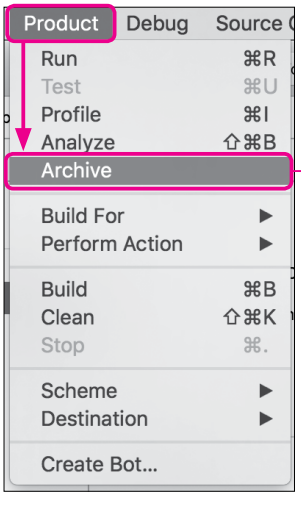
リリースするアプリはアーカイブ機能によって `ipa`というファイルにまとめる必要があります。そのファイルをiTunes Connectにアップロードします。

Archive

Xcodeでアプリをアーカイブします。



シミュレータではなく「Generic iOS Device」を選択します。

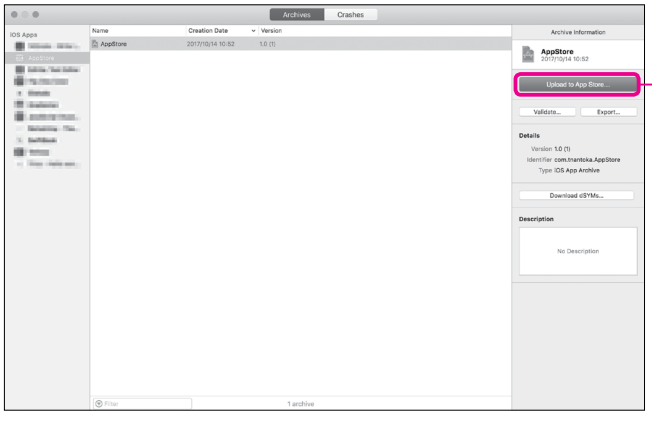


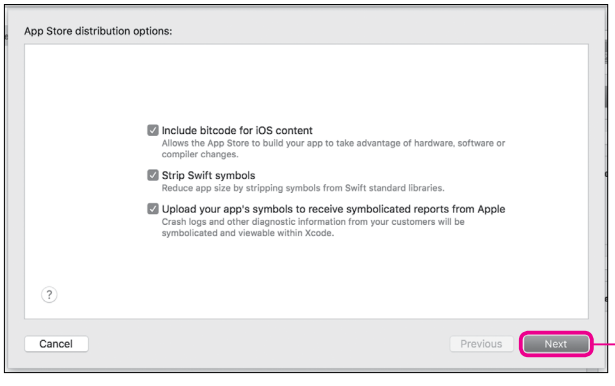
メニューバーから「Product」→「Archive」を選択します。

アーカイブが終わると Organizer というウィンドウが開きます。アップロード作業はこのウィンドウで行います。

Note

Organizerのウィンドウを閉じてしまった場合には、メニューバーから「Window」→「Organizer」を選択することで起動することができます。

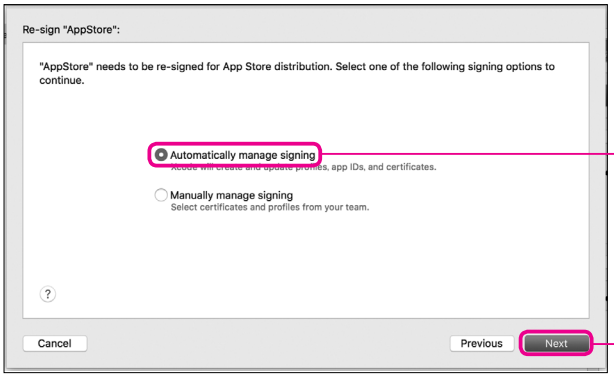




提出用の署名に使う証明書を作成する必要があります。この作業が必要なのは初回アップロード時のみです。

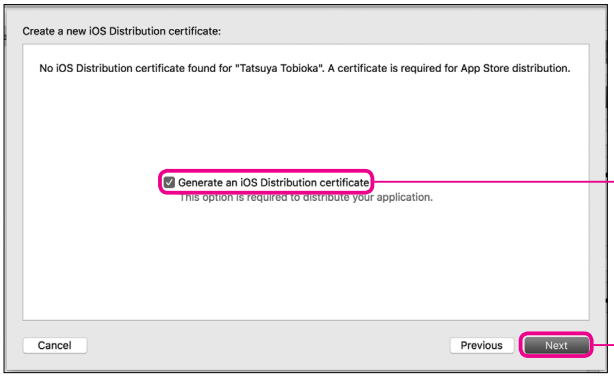
Note

書籍版のChapter 19の実機テストのときに作ったのは開発用の証明書で、こちらは本番用の証明書です。



①「Automatically manage signining」を選択します。

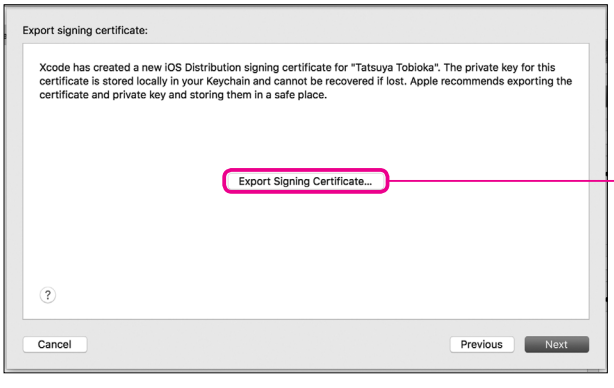
②「Next」をクリックします。



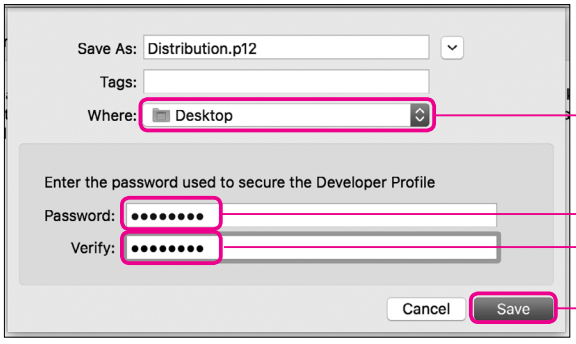
①「Generate an iOS Distribution certificate」を選択します。

②「Next」をクリックします。

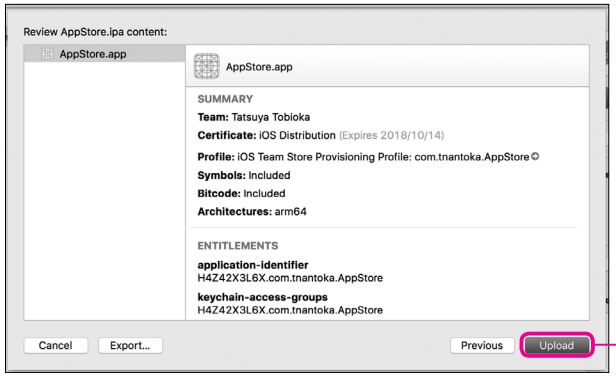
証明書を Export しておくことを勧められます。Export したファイルを他の Mac で開けば、その Mac でもリリース作業を行うことができます。Export してみます。



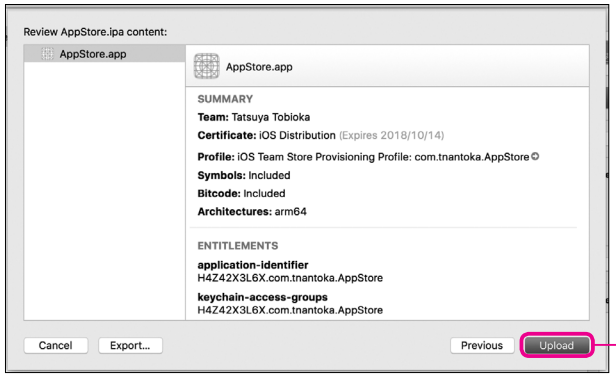
「Export Signing Certificate...」をクリックします。



あとはアップロードして待つだけです。



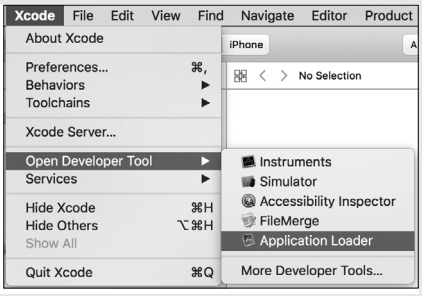
アップロードが成功すると、次のような画面が表示されます。



Note

「Validate...」というアップロード前にアーカイブを検証する機能もあります。しかし、たとえ検証の結果問題が見つからなくても、実際にアップロードするとエラーが発生することもあります。いきなりアップロードしてしまっても良いでしょう。

また、回線が不安定などの理由でアップロードが失敗する場合は、Xcodeに含まれる「Application Loader」というアプリを使うとうまくいくことがあります。「Application Loader」は、次のようにXcodeのメニューバーから起動することができます。



iTunes Connect上で確認・設定

アップロードが完了すると、iTunes Connectの「アクティビティ」にアップロードしたアプリが表示されます。

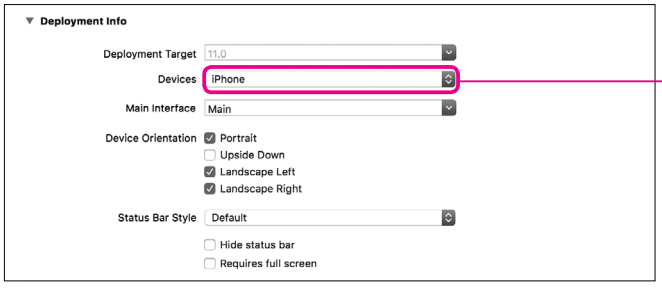
①「アクティビティ」を選択します。

②アップロードしたアプリの情報を確認します。

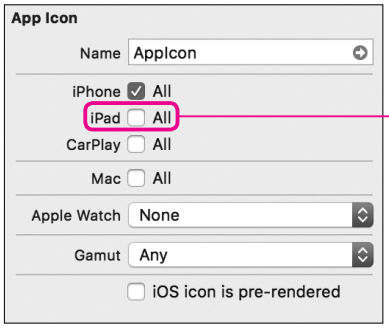
注意

「アクティビティ」にアップロードしたアプリが表示されない場合、何かエラーが起きている可能性があります。

例えばアイコンの設定を忘れているかもしれません。そのような場合には、Apple社からメールが届くはずです。アイコンを忘れていた場合は、以下のように対応します。ここではデフォルトのままユニバーサルにしてしまっていたので、iPhone専用アプリへの変更も行っています。

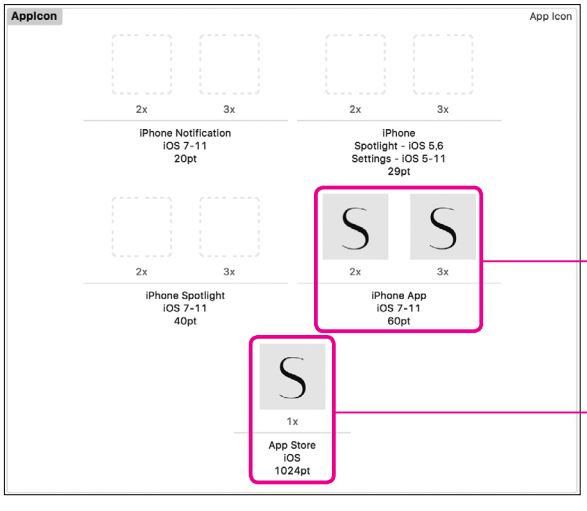


「Devices」で「iPhone」を選択します。



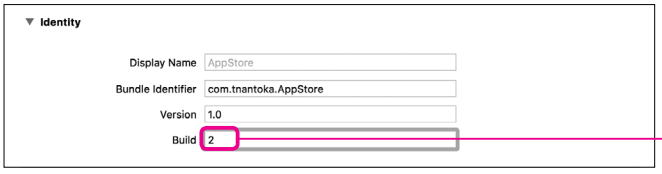
「App Icon」の「iPad」をオフにします。

アイコンを設定します。



Finderからドラッグ＆ドロップして、画像を設定します。

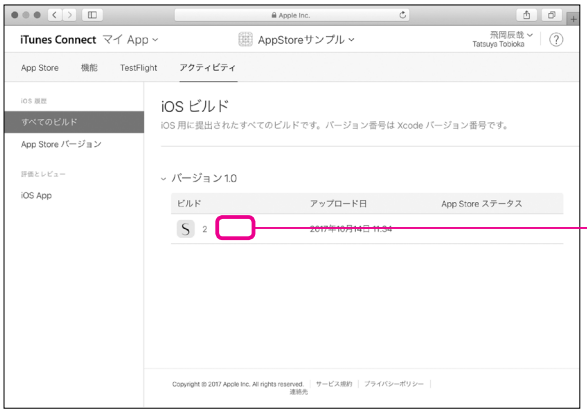
アプリをアップロードし直す場合、同じバージョン番号、かつ同じビルド番号ではアップロードすることができません。ここでは同じバージョンの再提出なのでビルド番号のみを変更します。



「Build」を2に変更します。

審査用のビルドを指定

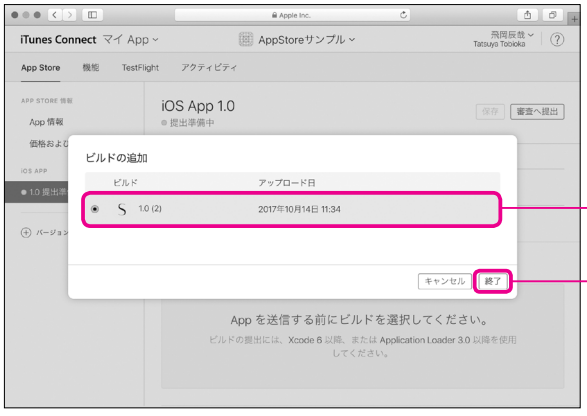
「アクティビティ」画面のアプリ情報から「処理中」という表示がなくなると、審査に進むことができます。処理が終わったときにもApple社からメールが届きます。



「処理中」の表示がなくなったことを確認します。

アップロードしたファイルを提出用に設定します。

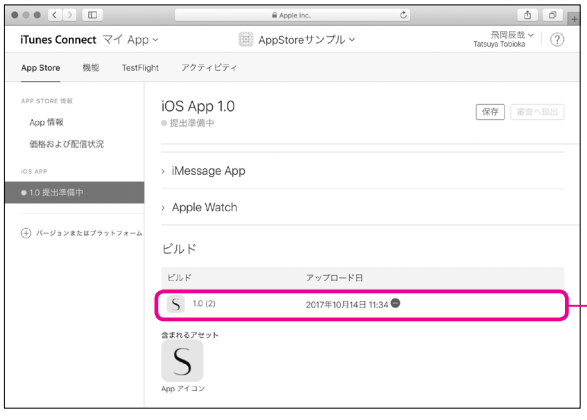
「ビルド」の右側の「＋」をクリックします。



①アップロードしたビルドを選択します。

②「終了」をクリックします。

アップロードしたビルドを選択して、「終了」をクリックします。



選択したビルドが表示されます。

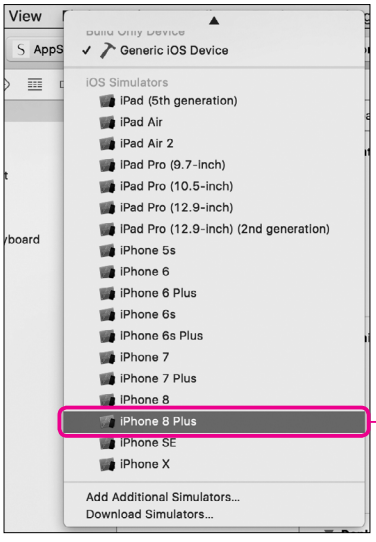
マウスカーソルを表示されたビルドにホバーすると表示される「−」（マイナス）をクリックすると取り消すことも可能です。

以上でArchiveの送信は終わりです。

5. スクリーンショットなどの登録

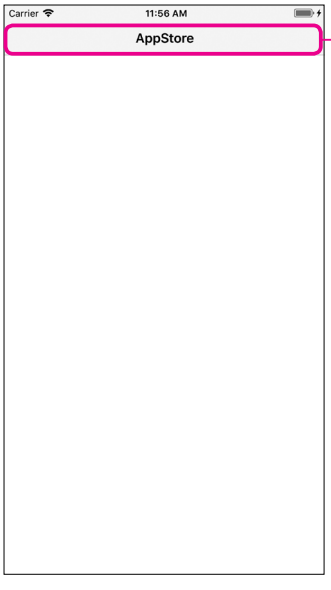
Archiveの送信が済んだら、あとは必要な情報をiTunes Connect上に登録すれば審査に提出することができます。

まず、アプリのスクリーンショットが1～10枚必要です。必要最低限で構わないのでしたら、現行モデルで最も画面サイズが大きい「iPhone 8 Plus」で撮ったスクリーンショットがあれば大丈夫です。



「iPhone 8 Plus」を選択します。

スクリーンショットを取得します。



真っ白だとわかりにくいので、ナビゲーションバーを表示しています。

スクリーンショットは、次の場所にドラッグ&ドロップすることでアップロードすることができます。

ここにドラッグ＆ドロップします。

「OK」をクリックします。

スクリーンショットがアップロードされます。

他の端末用にも自動でリサイズされて使用されます。

Note

各端末のスクリーンショットを個別に指定したい場合は、「メディアマネージャーですべてのサイズを表示」から追加でアップロードすることも可能です。

6. 審査に提出する

以上で全ての準備が整いました。最後に審査に必要な情報を入力して提出します。

①プライマリカテゴリを選択します。

②セカンダリカテゴリを選択します。

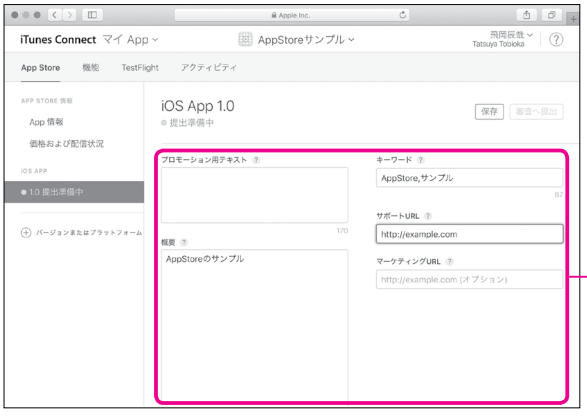
販売価格を設定します。

左側のメニューから「価格および配信状況」を選択します。

Note

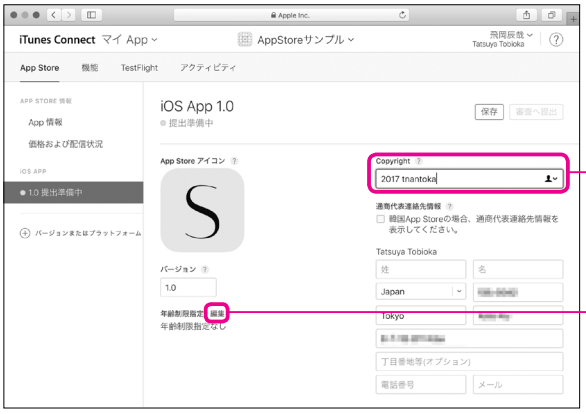
保存後に価格を変更したい場合は、「価格計画の変更」をクリックします。

「概要」「キーワード」「サポート URL」を入力します。「プロモーション用テキスト」や「マーケティング URL」は任意です。



AppStoreで表示されるアイコンをアップロードします。アプリ内でも指定した 1024pxのアイコンを「ファイルを選択」と表示されている場所へドラッグ＆ドロップすればOKです。

著作権情報（Copyright）と年齢制限指定を入力します。



①著作権情報を入力します。

②「年齢制限指定」の横の「編集」をクリックします。

ここでは「年齢制限指定」はすべて「なし」に設定します。

アプリ内でどのようなコンテンツが表示されるかなどの質問が並んでいます。

全て「なし」または「いいえ」を選択します。

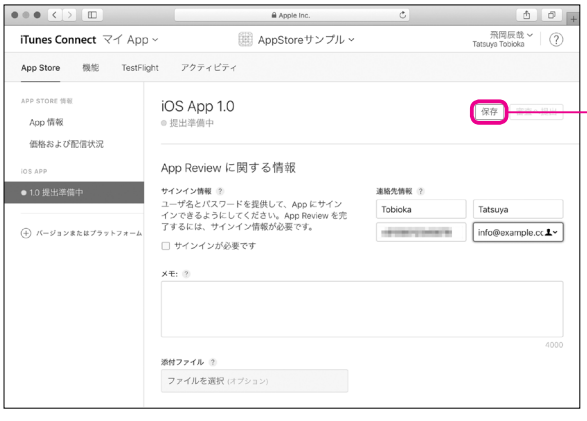
「終了」をクリックします。

「年齢制限指定」が「4＋」（4歳以上）と表示されていることを確認し、氏名や連絡先情報を入力します。

①「年齢制限指定」が「4＋」（4歳以上）と表示されていることを確認します。

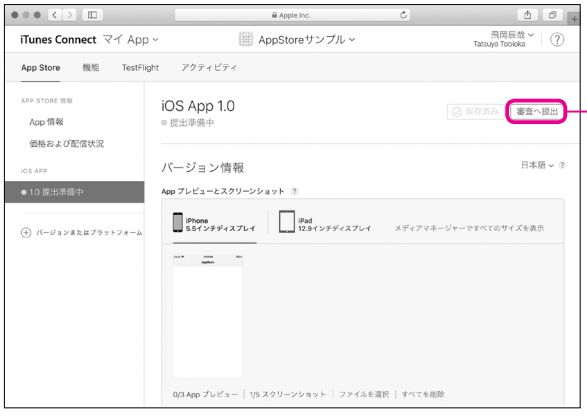
②氏名や連絡先情報を入力します。

入力が済んだら保存します。



「保存」をクリックします。

「審査へ提出」をクリックして、提出手続きを行います。アプリの内容に合わせて輸出コンプライアンス、コンテンツ配信権、広告IDに関する質問に回答してください。



「審査へ提出」をクリックします。

輸出コンプライアンスに関する質問を確認し、回答します。

①コンテンツ配信権、広告IDに関する質問を確認し、回答します。

②「送信」をクリックします。

元の画面に戻ります。画面左側のステータス表示が「審査待ち」に変わったことを確認してください。

「審査待ち」に変わったことを確認します。

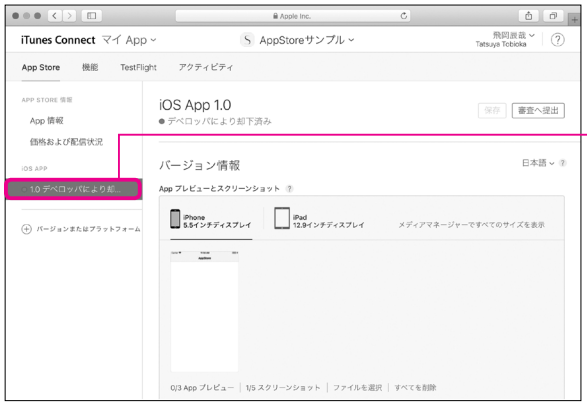
あとはApple社による審査の結果を待つのみです。

審査をキャンセルする

審査をキャンセルする場合は、「このバージョンを審査から削除」をクリックします。削除のリンクが表示されない時は、App情報など他のページにいったん移動したり、ログインし直したりしてみてください。

「このバージョンを審査から削除」をクリックします。

「削除」をクリックします。



画面のステータス表示が「デベロッパにより却下済み」に変わります。

Note

iTunes Connectの日本語版のヘルプが公開されています。困ったことがあれば参照してください。
『iTunes Connect デベロッパヘルプ』
http://help.apple.com/itunes-connect/developer/?lang=ja

Section 1.5 レビューガイドラインの注意点

iOSアプリを AppStoreで公開するためには、Apple社の審査を通過する必要があります。審査で違反と見なされる事項はApple社が公開するガイドラインに記載されています。

『App Store審査ガイドライン』
https://developer.apple.com/app-store/review/guidelines/jp/

例えば、次に挙げるような事由でリジェクトされることがあります。

- **クラッシュする。**
- **アプリ名に検索されやすいようにキーワードを入れている。**
- **何に使うかわからない情報が表示されている。**
- **ストレージガイドラインに違反している。(18章で紹介したディレクトリーを適切に使用しなかったなど)**
- **広告IDを広告表示に使用すると選択したのに、広告が表示されない。**
- **販売されているアプリのテンプレートを元に作成したアプリである。**
- **iPadのiPhoneモードでちゃんと動かない。**

「作ってから実は違反だった」などとならないように、常に最新のガイドラインを確認してください。

ここではアプリをストアにリリースする手順を紹介しました。開発したアプリを App Storeに公開する際の参考にしてください。

書籍本体、及び本PDFで紹介したWebサイトの一覧を以下にまとめます。

- 本書のサポートサイト
- サンプルプログラムのダウンロード

書籍

Chapter 1

- Apple Developer Programのサイト
- Apple社の開発者サイト

Chapter 2

- Apple Developer Documentation（英語）
- Stack Overflow（英語）
- Apple Developer Forums（英語）
- Qiita（日本語）
- teratail（日本語）

Chapter 3

- Apple Worldwide Developers Conference 2017資料

Chapter 7

- 「Stadiometer」ソースコード（ARKit活用例）

Chapter 8

- 「Kickstarter」ソースコード
- 「Gradientor」ソースコード
- 「edhita」ソースコード
- Apple社サンプルコード

Chapter 11

- SpriteKit.jp：Sprite Kit 日本語情報サイト

Chapter 12

- UIKitの公式ドキュメント（UIButton）
- UIKitの公式ドキュメント（UIControlEvents）

Chapter 13

- Apple Human Interface Guidelines/Overview/iPhone X

Chapter 15

- Core Image Filter Reference

ボーナスPDF

Chapter 1

- Icon Creator
- Apple Human Interface Guidelines
- GitHub Pages
- Netlify
- WordPress.com
- Apple開発者用サイト
- iTunes Connect
- iTunes Connect デベロッパヘルプ
- App Store審査ガイドライン

Chapter 2

- App Store審査ガイドライン
- Apple技術資料「Technical Q&A QA1689 Supporting orientations for iPad apps」
- Apple Human Interface Guidelines/Visual Design Layout

Chapter 4

- CocoaPods
- SPM（Swift Package Manager）
- Carthage/README.md
- Carthageリリース一覧
- Alamofire
- PKHUD
- Eureka
- SwiftIconFont
- FontAwewome
- KeychainAccess
- Realm
- Realm日本語ドキュメント
- LicensePlist
- LicensePlistリリース一覧
- The MIT License
- Apache License, Version 2.0
- LicensePlist Settings.bundle
- SwiftLint
- SwiftLintリリース一覧

『Swift 4プログラミング入門』ボーナスPDF

Chapter 2

iPadに対応させる

iPadは、iPhone専用のアプリでも、拡大表示をして使うことができます。しかし、iPadに対応したアプリの方が使いやすいのは間違いありません。本PDFの Chapter 2では iPad対応の方法や注意点について紹介します。

Section 2.1

iPad対応をする理由

App Store審査ガイドライン（https://developer.apple.com/app-store/review/guidelines/jp/）に以下の記述があります。

> ユーザーがアプリケーションを最大限に活用できるよう、iPhone向けのアプリケーションは可能な限り iPadでも実行できるようにしてください。Appleは、ユーザーがすべてのデバイスで使用できるように、ユニバーサルアプリケーションの構築を検討するようおすすめしています。

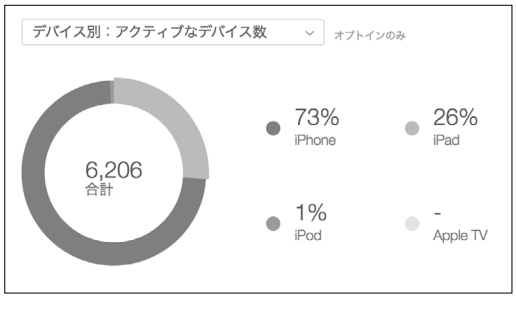
iPad対応していないことを理由に審査がリジェクトされる可能性は低いですが、やはり Appleとしては iPadでも動くようにしてほしいという思いがあるようです。また、iPadユーザーにとってもダウンロードしたアプリがiPadにも対応しているとやはり嬉しいものです。

ただ、全てのアプリが iPad対応の手間をかけられるわけではないですし、そもそも iPadに向かないアプリもあるでしょう。対応の余地があるアプリであれば是非検討してみてください。

アプリの絶対数がiPhoneよりも少ないので、ランキングに入りやすい可能性もあるでしょう。

Note

参考までに、以下はあるアプリの2017年11月のアクティブなデバイス数です。



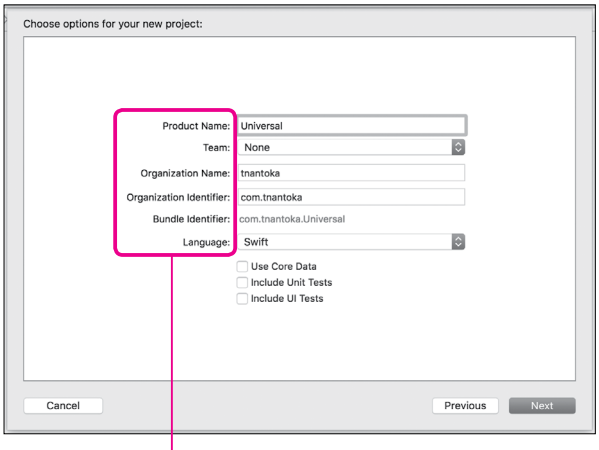
約4分の1のユーザーがiPadを使っており、このアプリはiPad対応する価値があったと言えるでしょう。

Section 2.2

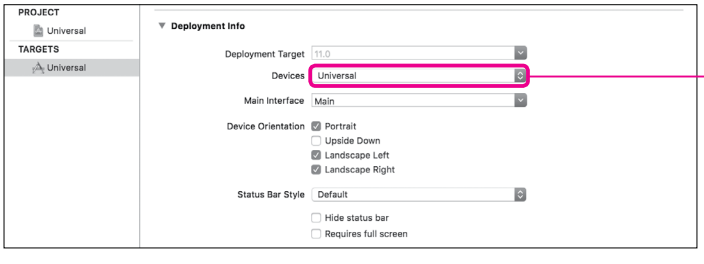
iPad対応プロジェクト

iPhoneと iPadを大きく見た目を変える場合、かなり手間がかかりますが、そのままの見た目で対応する場合はオートレイアウトの恩恵もあり、とても簡単です。Storyboardファイルで、デバイスの大きさが変わっても適切に表示されるようAutoLayoutを設定しておけば、iPadでもうまく表示させることができます。

Xcode 9以降で新たに作るアプリであれば、デフォルトで iPad対応の `Universal`（ユニバーサル）アプリになっています。

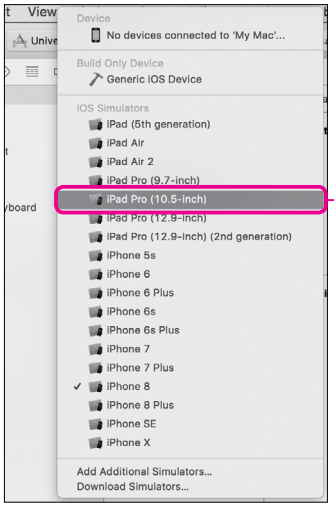


Xcode 8以前ではデバイス選択欄がありましたが、Xcode 9ではデバイス選択欄がなくなり、デフォルトで「Universal」になっています。

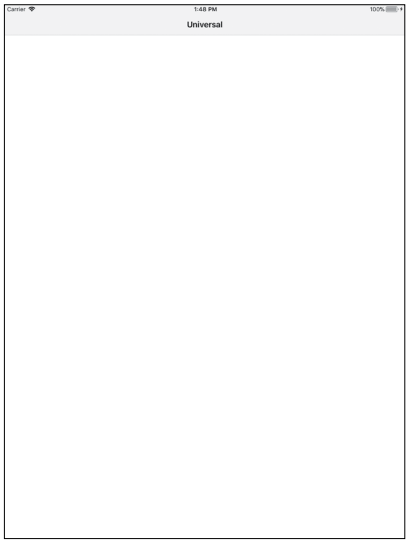


「Deployment Info」の「Devices」欄もデフォルトで「Universal」になっています。

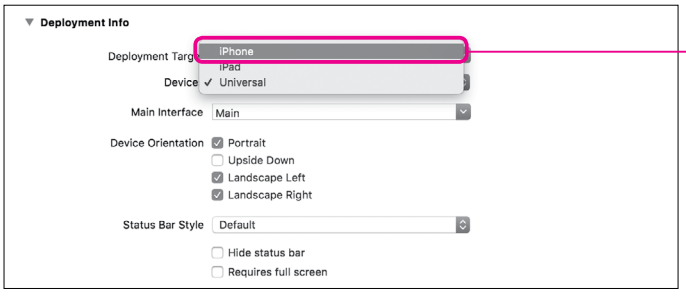
「iPad Pro (10.5-inch)」を選んでアプリを実行してみます。



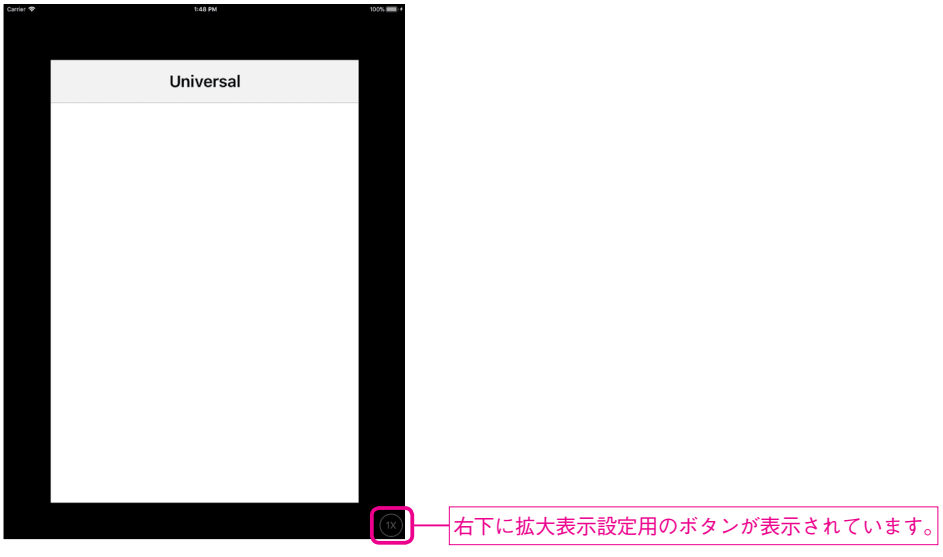
これだけでiPad対応のアプリとなります。



iPhoneだけに対応したい場合には「Deployment Info」の「Deployment Target」を「iPhone」に変更します。



設定変更後に再度アプリを実行すると、iPhoneモードで実行されます。



Note

もしiPhone対応が不要であれば、iPadを選べばiPad専用のアプリになります。

Section 2.3

iPadアプリに必要なアイコン

iPadアプリの開発には、以下のようなサイズのアイコンが必要です。

サイズ	必須
20px	−
29px	−
40px	−

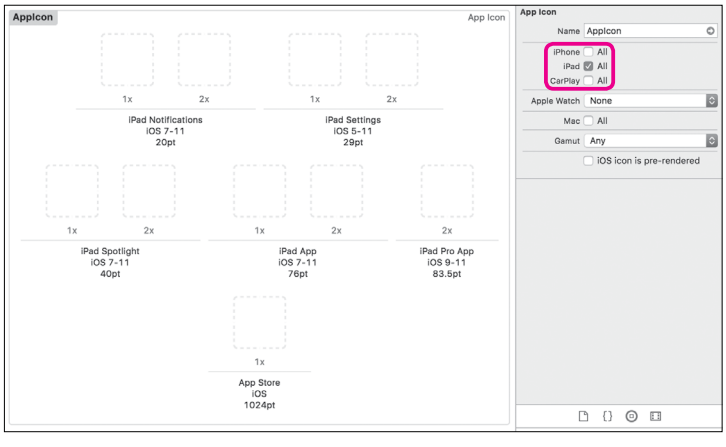
サイズ	必須
58px	−
76px	○
80px	−

サイズ	必須
152px	○
167px	○
1024px	○（Xcode 9から）

本 PDFの Chapter 1で紹介した iPhoneアプリで必要とされるアイコンサイズと一部重複しています。ユニバーサルアプリの場合、どちらのサイズのアイコンも必要です。

Assets.xcassetsで iPadのみをチェックした状態にすると、以下のように iPadに必要なアイコンの設定だけが表示されます。

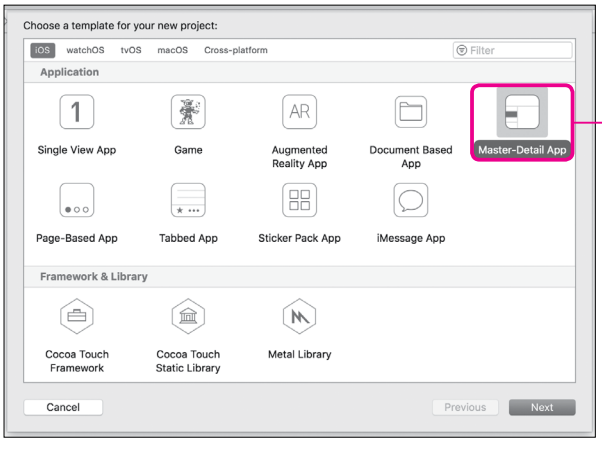
設定箇所は10箇所ありますが、40pxが2箇所あるため全部で9種類になります。



Section 2.4

Split View Controller

書籍版の Chapter 1で紹介した「Master-Detail App」テンプレートを使ったアプリでは **UISplitView Controller**というコンポーネントが使われます。



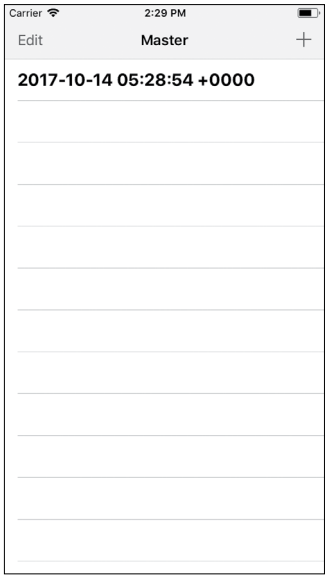
プロジェクト作成時に「Master-Detail App」を選択します。

「Master-Detail App」テンプレートを使ったアプリは、デバイスに応じて次のように表示が変化します。

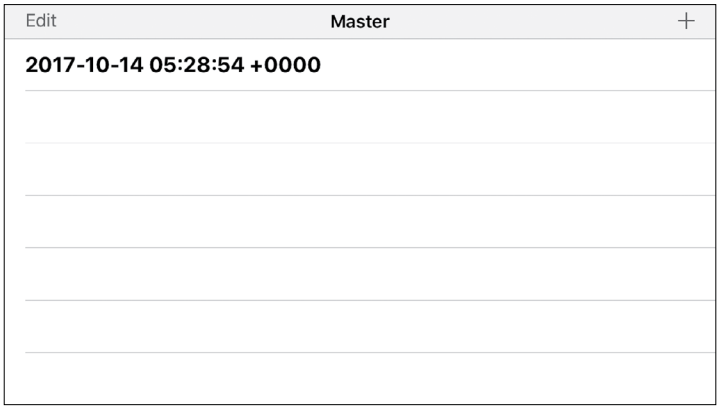
デバイス	方向	表示
iPhone	Portrait	一覧から詳細に遷移していく表示。
	Landscape	一覧から詳細に遷移していく表示。
iPhone Plus	Portrait	一覧から詳細に遷移していく表示。
	Landscape	2ペインに別れた表示。
iPad	Portrait	詳細画面に一覧が重なる表示。
	Landscape	2ペインに別れた表示。

iPhone 8の画面

Portrait

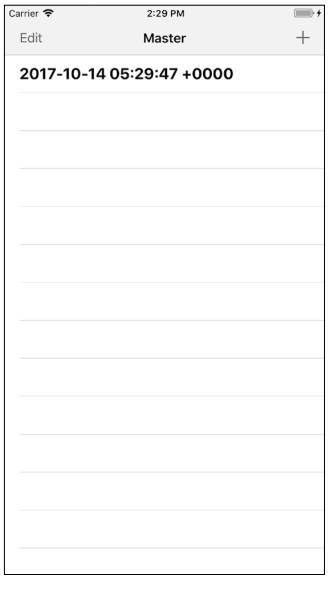


Landscape

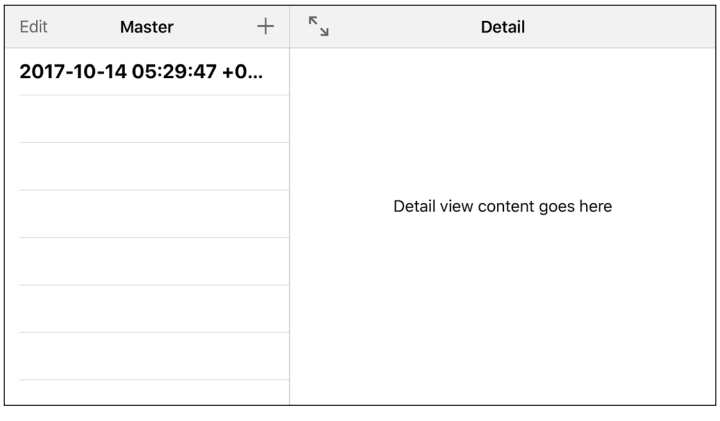


iPhone 8 Plusの画面

Portrait

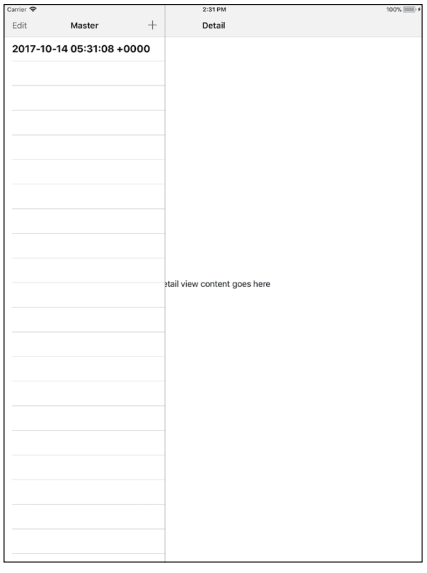


Landscape

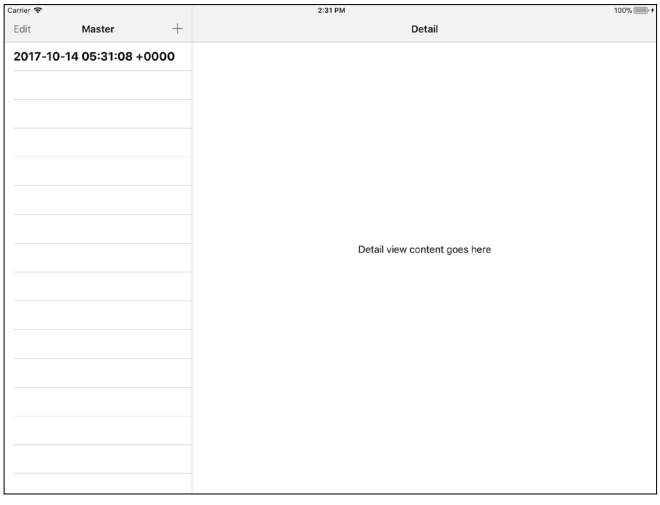


iPad Pro(10.5-inch)の画面

Portrait



Landscape



データを一覧表示するようなアプリの場合、「Master-Detail App」テンプレートを使うことで、iPhone、iPadに対応したユニバーサルアプリを簡単に開発することができるでしょう。

Note

例えば、標準アプリの「メール」や「メモ」「設定」のようなアプリを作りたい場合、`UISplitViewController`の使用を検討するとよいでしょう。

Section 2.5

iPad対応の注意点

アプリをiPad対応するときの注意点です。

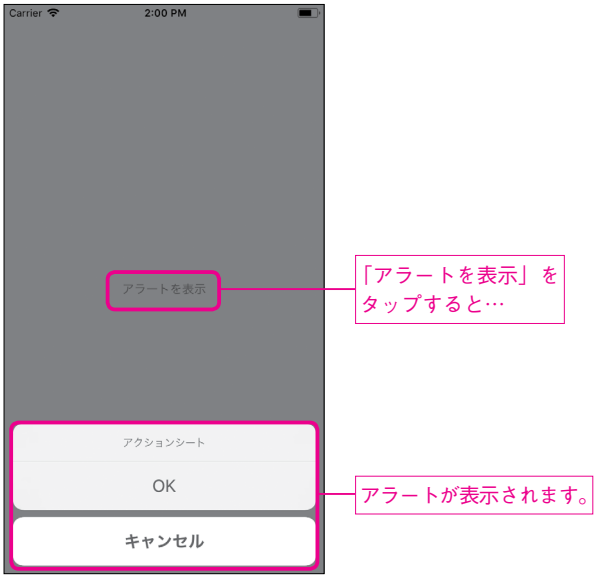
iPhone向けのコードがクラッシュ

例えば、書籍版の Chapter 13で紹介したアクションシートのサンプルプログラムは iPadで実行するとクラッシュしてしまいます。

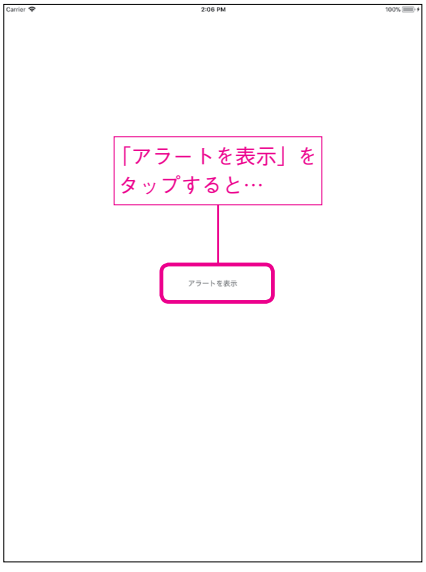
サンプルプログラム　Bonus/Chapter2/Popover/Popover/ViewController.swift

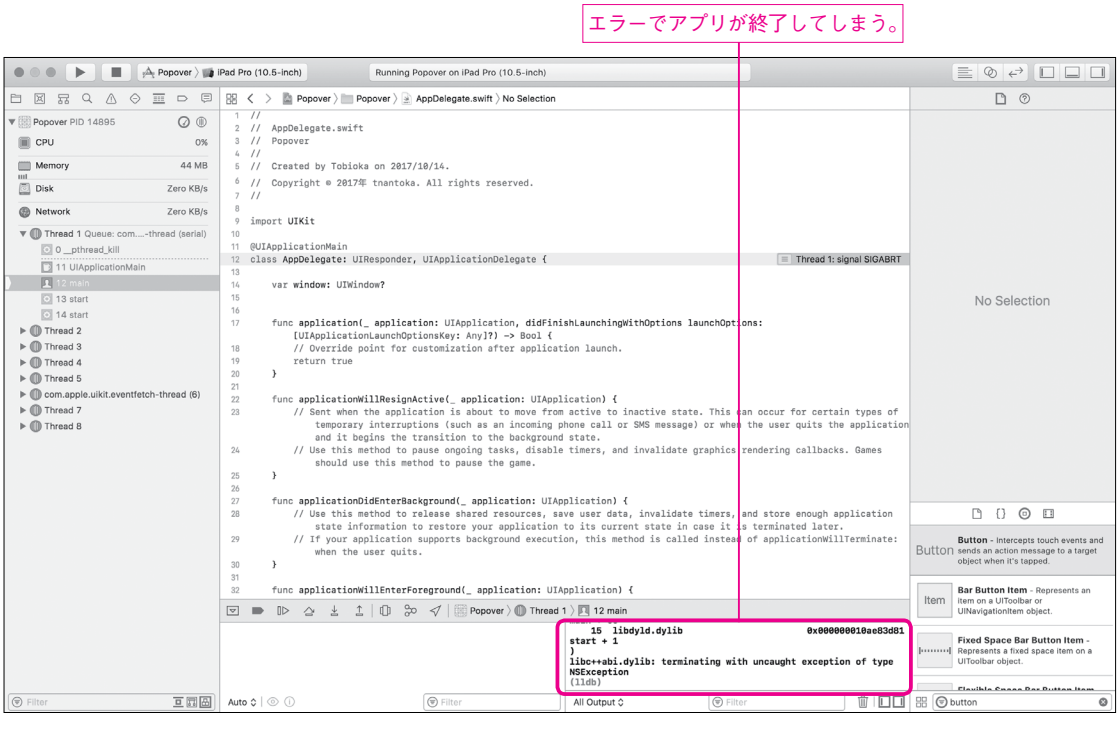
```
let alertController = UIAlertController(
    title: "アクションシート",
    message: nil,
    preferredStyle: .actionSheet
)

alertController.addAction(
    UIAlertAction(
        title: "キャンセル",
        style: .cancel,
        handler: nil
    )
)

alertController.addAction(
    UIAlertAction(
        title: "OK",
        style: .default,
        handler: nil
    )
)

present(alertController, animated: true, completion: nil)
```

iPhoneでは問題ない



iPadではクラッシュ



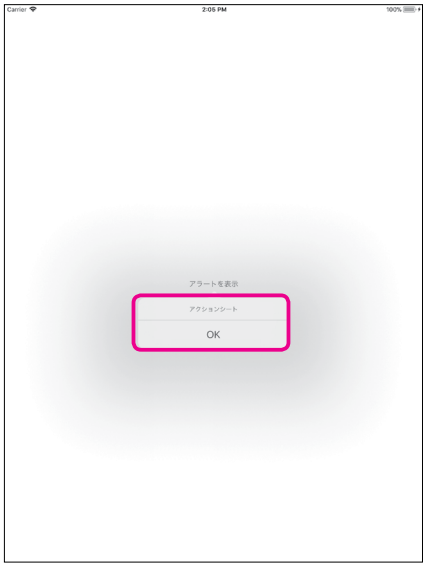


原因は、**iPadではActionSheetをポップオーバーとして表示させる**必要があるためです。

サンプルプログラム　Bonus/Chapter2/Popover/Popover/ViewController.swift

```
alertController.popoverPresentationController?.sourceView = sender
alertController.popoverPresentationController?.sourceRect = sender.bounds
alertController.popoverPresentationController?.permittedArrowDirections =
[.up]
```

iPadではポップオーバーで動作



Note

ActionSheetをポップオーバーで表示させても、iPhoneでの実行に影響はありません。

画面方向

iPadアプリは基本的に全画面方向で使えることが望ましいとされていました。

例えば 2010 年に Apple 社が公開した技術資料『Technical Q&A QA1689 Supporting orientations for iPad apps』(https://developer.apple.com/library/content/qa/qa1689/_index.html) には、次のような記述があります。

> It is strongly recommended that your application support all orientations.（全画面方向をサポートすることを強く推奨します。）

一方、現行の『ヒューマンインターフェースガイドライン』（https://developer.apple.com/ios/human-interface-guidelines/visual-design/layout/）にはiPadに限定した記述は存在しません。しかし、「可能であれば縦・横両方向サポートすること」という記述があります。

> If possible, support both portrait and landscape orientations.（可能であれば縦・横両方の画面方向をサポートしてください。）

現行のiPadはマルチタスク機能をサポートしており、2つのアプリを並べて使うことができます。マルチタスク機能に対応するためには、全画面方向のサポートが必要です。

Xcode 9で作成するプロジェクトは、デフォルトの状態でマルチタスク機能をサポートする設定になっています。

マルチタスク機能をサポートしない場合は「Deployment Info」の「Requires full screen」をチェックします。

デフォルトでは「Requires full screen」はオフになっています。

やはり、基本的にはiPadアプリは全画面方向をサポートすべきと考えた方がよいでしょう。

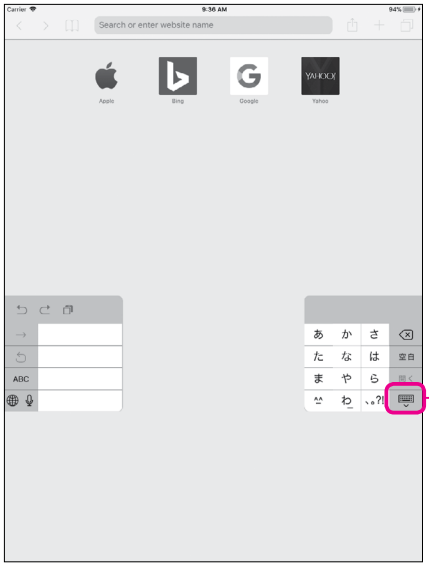
もともと全画面方向に対応していたアプリなら、iPad対応してしまった方が良いという考え方もできます。

また、もともと1方向限定だったアプリを別方向に対応する場合は、崩れがないか入念に確認しましょう。

キーボード

キーボードに関しても注意しておく必要があります。

iPadではハードウェアキーボードがiPhoneに比べてよく使われるほか、キーボードを移動・分割するモードが提供されています。



iPadでは右下のキーボードキーを長押しすることで、モードを切り替えることができる。

文字入力が重要なアプリでは、テストを欠かさないようにしましょう。

この他の箇所も予期せぬ動作をする可能性があります。単純にユニバーサル化しただけのアプリでも、必ず実機での動作を確認しましょう。

ここでは iPad対応について簡単に紹介しました。 特に、同じ画面構成で iPadでも動けば便利そうなアプリなら、是非ユニバーサル対応してみてください。

『Swift 4プログラミング入門』ボーナスPDF

Chapter 3

Xcodeの詳しい使い方

書籍版のChapter 1では数あるXcodeの機能のうち、本書を進めるための最低限のものだけを紹介しました。Xcodeには他にもたくさんの機能があります。ここでは開発の効率化に役立ちそうなものをピックアップして紹介します。

Section 3.1

Playground

今までは iOSアプリのプロジェクトを使ってきましたが、Playgroundという気軽に Swiftのコードを書いて試せる機能もあります。断片的なコードを試したいときなどに便利です。

Playgroundの作成

プロジェクトと同じく、Xcode起動時に表示される Welcome to Xcode画面から作成できます。また、Xcodeのプロジェクト内に追加することも可能です。

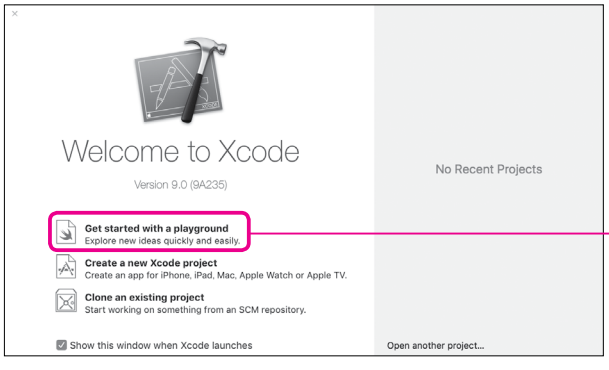
ここではWelcome to Xcode画面から単体のPlaygroundを作成します。

サンプルファイル

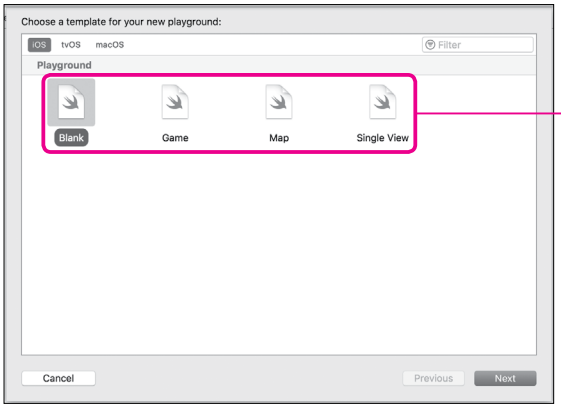
種類	ファイル
Playground	Bonus/Chapter3/Playground/Blank.playground

Xcode 9から以下のようなテンプレートを利用できるようになりました。

テンプレート	概要
Blank	文字列の変数が１つ定義されているだけのシンプルなテンプレート。
Game	SpriteKitを使ったゲームのテンプレート。
Map	MapKitを使った地図のテンプレート。
Single View	ラベルをもつ単純なビューを表示するテンプレート。



「Get started with a playground」をクリックします。



Note

テンプレート一覧画面は、Xcodeのメニューバーから「File」→「New」→「Playground」を選択することでも表示できます。

ここでは「BLANK」テンプレートを選んで、Playgroundを作成します。「Next」をクリックすると保存場所を尋ねられるので、適当な場所を選び、「Create」をクリックします。

実行

テンプレートの名前は「BLANK」ですが、実際には次のようなプログラムが入力されています。

```
//: Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

画面上部のプログレスインジケーターを確認して、シミュレーターが起動する（「Ready」と表示される）までしばらく待ちます。

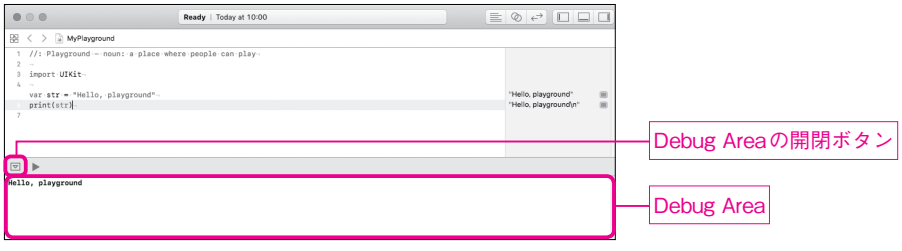
シミュレーターが起動すると、画面右側に "Hello, playground" という変数 `str` の中身が表示されます。以降もコードを編集たびに自動的に表示が更新されていきます。

print

これまでのプロジェクトと同じく `print` を使うと「Debug Area（デバッグエリア）」で出力結果を確認できます。入力されているプログラムの末尾に、次のプログラムを追加します。すると、画面下部に「Debug Area」が表示されます。

サンプルプログラム　Bonus/Chapter3/Playground/Blank.playground/Contents.swift

```
print(str)
```



ビュー

liveViewという機能を使うと、「Assistant Editor（アシスタントエディター）」でビューを確認することができます。

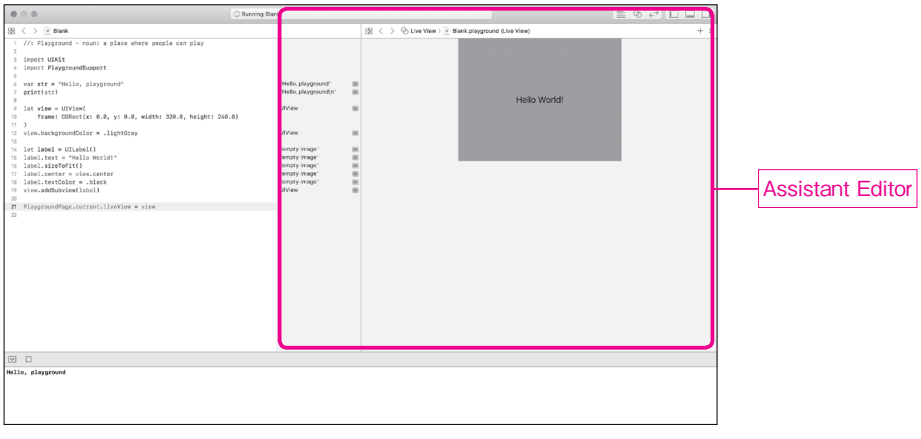
入力されているプログラムをすべて削除して、次のプログラムを入力してください。このプログラムでは単色の`UIView`を表示しています。

サンプルプログラム　Bonus/Chapter3/Playground/Blank.playground/Contents.swift

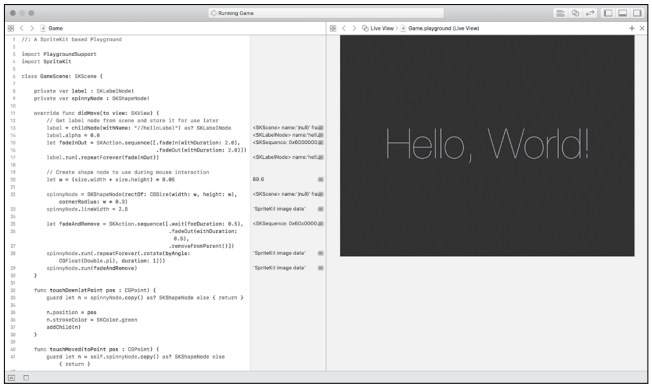
```
import UIKit
import PlaygroundSupport

// グレーのビューを作成
let view = UIView(
    frame: CGRect(x: 0.0, y: 0.0, width: 320.0, height: 240.0)
```

```
)
view.backgroundColor = .lightGray

// ラベルを作成してビューに追加
let label = UILabel()
label.text = "Hello World!"
label.sizeToFit()
label.center = view.center
label.textColor = .black
view.addSubview(label)

// ビューを表示
PlaygroundPage.current.liveView = view
```

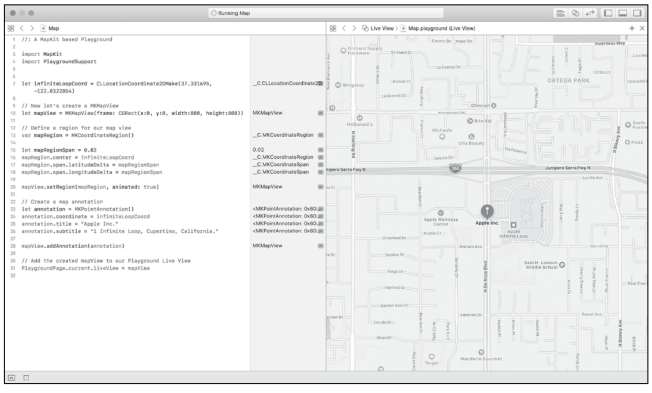
サンプルプログラムを入力後、メニューバーから「View」→「Assistant Editor」→「ShowAssistant Editor」を選択するとAssistant Editorが表示されます。



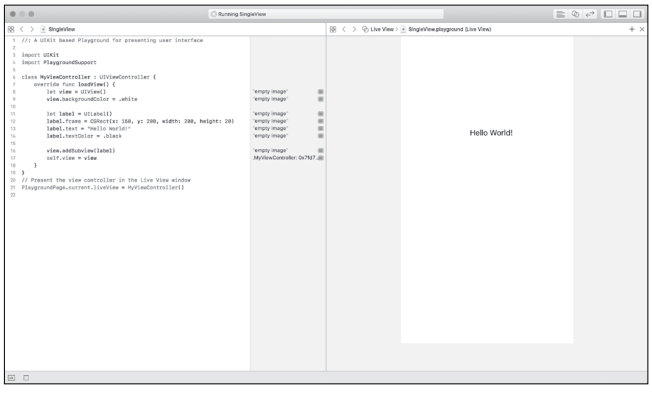
GameやMap、Singe Viewテンプレートを使う場合には、基本的にliveViewを使うことになります。



「Game」テンプレートの実行画面



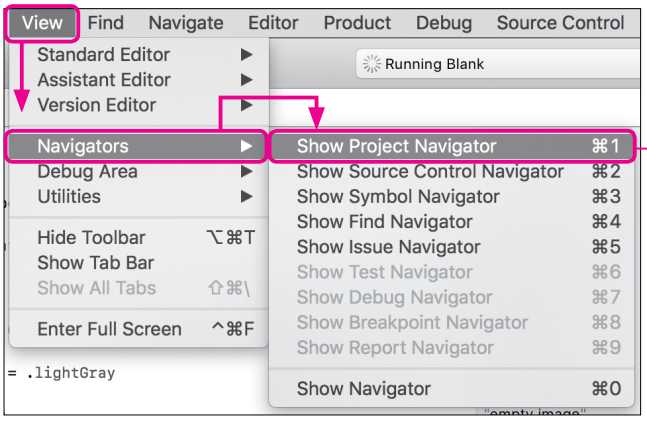
「Map」テンプレートの実行画面



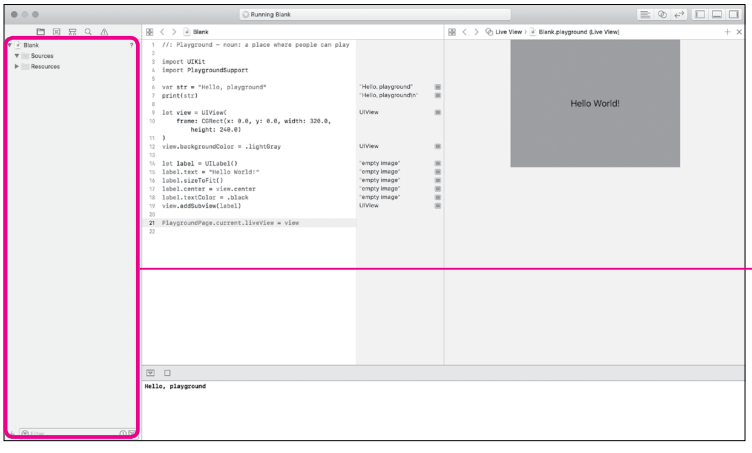
「Single View」テンプレートの実行画面

Project Navigator

通常の Xcodeプロジェクトと同じく、Playgroundでも「Project Navigator」を表示させることができます。ファイルや画像を追加する場合は「Project Navigator」を使います。



メニューバーから「View」→「Navigators」→「Show Project Navigator」を選択します。



「Project Navigator」が表示されています。

例えば独自のクラスをPlaygroundで使いたい場合は、「Project Navigator」の「Sources」に`.swift`ファイルを追加し、その中にクラスを書くことで実現できます。ただし、`public`か `open`なクラスとして定義しないとPlaygroundから呼び出せないため注意してください。

また、画像などを使いたい場合は「Resouces」に追加します。その画像は`UIImage(named:)`などの方法で読み込むことができます。

このようにビルドしたり、シミュレーターを起動したりすることなく、書いたプログラムを手軽に試すことができます。

Note

書籍版のPart 2で掲載しているサンプルプログラムはPlaygournd でも動作を検証しています。
サンプルプログラムのPlaygroundフォルダーの中に入っていますので、必要に応じてご利用ください。

Section 3.2

変数名の一括変更

Xcodeには変数名などの変更を一度に行える**リファクタリング**機能がついています。複数ファイルをプレビューしながら行えるので便利です。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter3/Refactor.xcodeproj
Storyboard	変更しません
ソース	Bonus/Chapter3/Refactor/Refactor/ViewController.swift

`ViewController.swift`内に、定数を1つ`print`するコードを書きます。

サンプルプログラム　Bonus/Chapter3/Refactor/Refactor/ViewController.swift

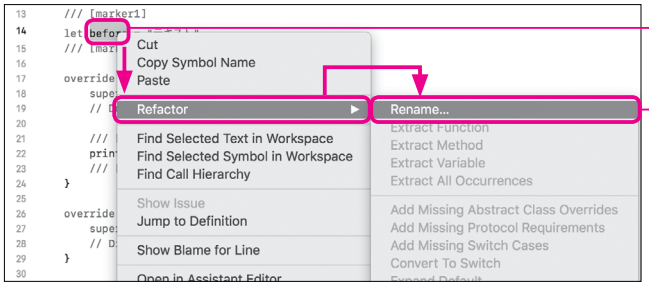
```
let before = "テキスト"
print(before)
```

続いて、別のファイルから、この定数を使います。

サンプルプログラム　Bonus/Chapter3/Refactor/Refactor/AppDelegate.swift

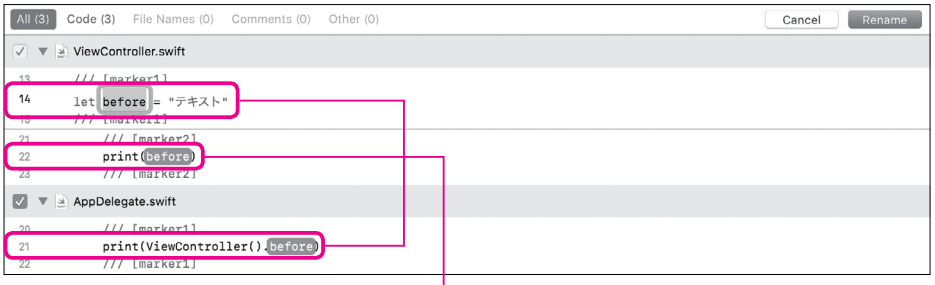
```
print(ViewController().before)
```

それでは、リファクタリング機能を使って定数名を変更してみます。

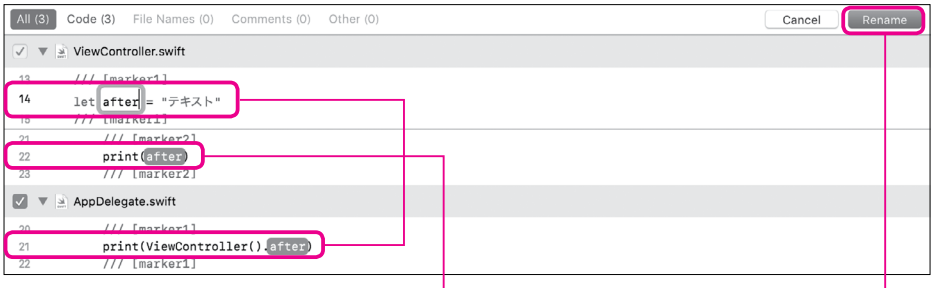


①変更したい定数（ここではViewController.swiftの方）を選択して、右クリックします。

②表示されたメニューで「Refactor」→「Rename」を選択します。



選択した定数を使用しているファイルが開き、定数が使われている箇所が表示される。



①「after」に変更すると、該当箇所がすべて自動的に変更されます。

②右上の「Rename」ボタンをクリックすると変更が確定されます。

このように一括でプレビューしながら変更できます。

Note

リファクタリング機能は Objective-C 向けとしてはバージョン 8以前の Xcode でも提供されていました。Swift には長らく対応していませんでしたが Xcode 9でついに対応されました。とても便利な機能ですので、ぜひ活用してください。

Section 3.3

便利なキーボードショートカット

Apple社はXcode Keyboard Shortcuts and Gesturesという文書を公開しています。この文書にはXcodeで利用できるショートカットが多数記載されていますが、覚えきれる量ではありません。また、この文書は2011年から更新されていません。ここでは有用と思われるショートカットを厳選して紹介します。

以降で紹介するショートカットは、2018年3月にXcode 9.0での動作を確認しています。

Xcode

操作	キー
実行	command＋R
ビルド	command＋B
終了	command＋.
テスト	command＋U
コメントアウト・アンコメント	command＋/
クイックオープン	command＋shift＋O
矩形選択	option＋マウス
行の先頭にカーソル移動	control＋A
行の末尾にカーソル移動	control＋E
行選択	shift＋↑/shift＋↓
検索	command＋F
自動インデント整理	control＋I
アシスタントエディターに切り替え	option＋command＋return
標準エディターに戻す	command＋return

ビルドはアプリの実行はせずにソースコードのコンパイルのみ行う機能です。

クイックオープンは、以下のようにファイル名の部分一致などを元に、素早く目的のソースコードなどを開くことができる機能です。

矩形選択は複数行の同じ列を一度にコピー＆ペーストしたりできる機能です。

```
18  // ラベルを作成してビューに追加
19  let label = UILabel()
20  label.text = "Hello World!"
21  label.sizeToFit()
22  label.center = view.center
23  label.textColor = .black
24  view.addSubview(label)
```

iOSシミュレーター

操作	キー
デバイス回転	command＋←/command＋→
ホームボタン	command＋shift＋H
スクリーンショット	command＋S
ハードウェアキーボード有効・無効切り替え	command＋shift＋K
ソフトウェアキーボード表示・非表示切り替え	command＋shift＋K

ここで紹介したショートカットキーだけでも使いこなせれば、作業効率が向上するはずです。

Section 3.4

デバッグの方法

プログラムがうまく動かない時に、printを使って変数の中身などを確認することがあります。この手法はprintデバッグと呼ばれ、手軽に使えて便利ですが、コードに変更を加えているためあまり良い方法とはいえません。

理論的にはprintを追加したことでプログラムの処理が変わってしまい、確認したいバグが再現（発生）しなくなってしまう可能性もあります。また、printを消し忘れてデバイスのログに大事な情報を出力し続けてしまう可能性もあります。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter3/Debug.xcodeproj
Storyboard	Bonus/Chapter3/Debug/Debug/Base.lproj/Main.storyboard
ソース	Bonus/Chapter3/Debug/Debug/ViewController.swift

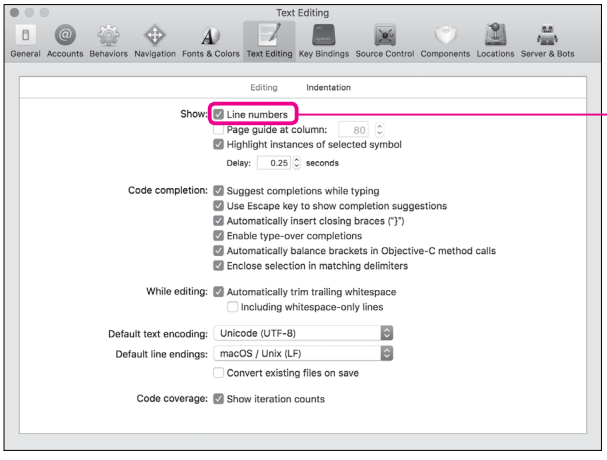
ログ出力

Xcodeには高機能なデバッガーが付属しているため、実行中のプログラムで使われている変数の値を見るのは簡単です。

ここでは、筆者が用意したサンプルプログラム（画面をタッチしたら count変数をインクリメントするアプリ）を使ってデバッグやログ出力の方法を説明します。

プログラムの編集時に行番号を表示させる

Editorでサンプルプログラムを開き、行番号が表示されていない場合は、Xcodeのメニューバーから「Xcode」→「Preferences」を選択し、「Text Editing」画面の「Editing」タブで「Show:」グループの「Line Numbers」をオンにします。行番号がなくても操作はできますが、わかりづらいので表示しておく方がよいでしょう。



「Line Numbers」をオンにします。

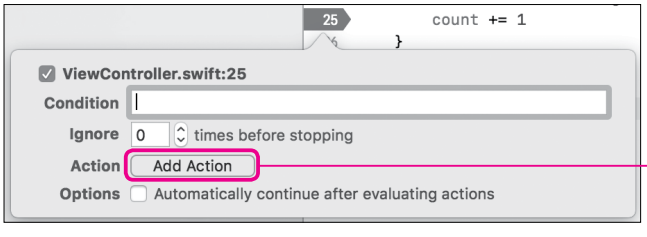
ブレークポイントを設定する

行番号部分をクリックするとブレークポイントが設定されます。
ここでは`ViewController.swift`の25行目をクリックしてブレークポイントを設定します。

```
24      override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
25          count += 1
26      }
```

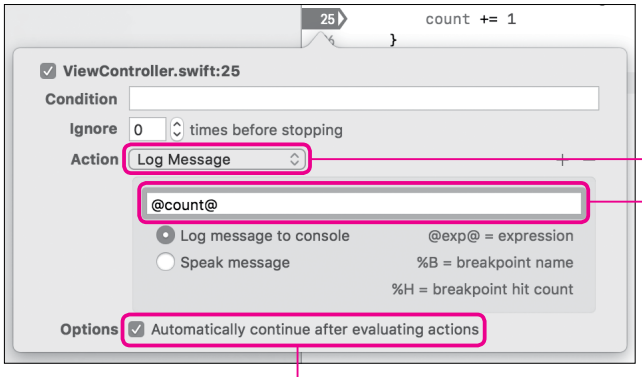
行番号をクリックします。

色がついた行番号をダブルクリックすると、編集画面が開きます。「Add Action」をクリックします。



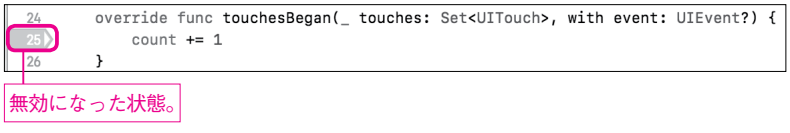
「Add Action」をクリックします。

count変数の中身をログに出力します。



この状態でアプリを実行すると、変数の内容がDebug AreaのConsoleへ出力されます。

ブレークポイントは行番号のクリックで無効・有効を切り替えられます。また、行番号部分の外側にドラッグ&ドロップすると削除できます。



少し難しそうに見えますが、慣れれば簡単にできるようになります。

対話的な操作

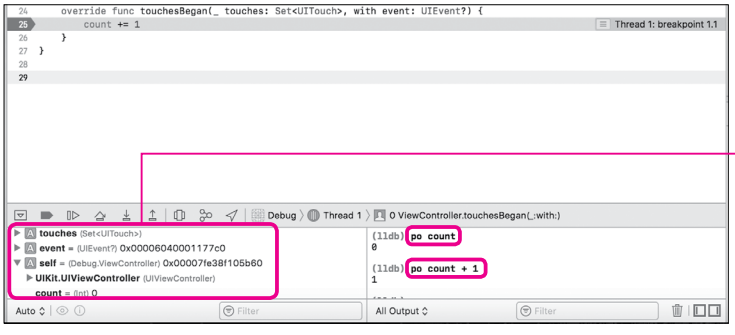
ブレークポイントを設定することで、プログラムを一時停止して対話的に操作することもできます。
ブレークポイントを設定し、何もせずにそのままアプリを実行してみましょう。

```
24      override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
25          count += 1
26      }
```

行番号をクリックするだけです。

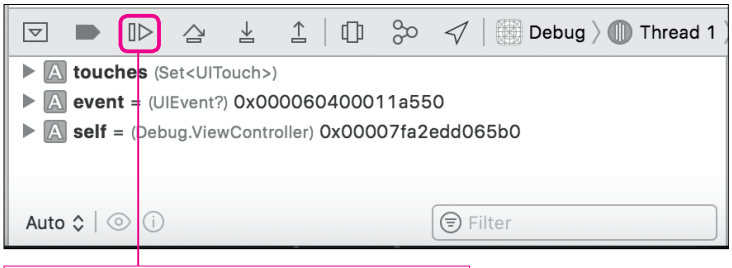
すると、アプリがそこでストップし、「Debug Area」が表示されます。「Debug Area」の左には変数などが表示されており、中身を確認することができます。

また、「Console」入力可能な状態になっていて、ここで「po 変数名」と入力すると変数の内容が表示されます。変数名の部分はSwiftのプログラムなので、例えば変換して表示するなどもできます。画面の例では`count`に`+ 1`しています。



変数やその中身が表示されます

「Continue program execution」をクリックするとアプリの処理が復帰します。



「Continue program execution」をクリックします。

デバッガーには他にも様々な機能があります。

まずは`print`の代わりに使うことから始めて、少しずつその他の機能も試してみてください。

Section 3.5

特殊なコメント

書籍版の Section 2.4「コメント」で、「コメントの書式に沿って書かれた箇所はプログラムの実行時に無視されるため、処理に影響を与えることはありません」と説明しました。しかし、Xcodeを使って開発するときだけ特殊な意味を持つコメントがあります。

こうした特殊なコメントを使うことで、Xcodeがより便利になります。

サンプルファイル

作りかけのカウンターアプリのプログラムを例として使います。

種類	ファイル
プロジェクト	Bonus/Chapter3/Comment.xcodeproj
Storyboard	Bonus/Chapter3/Comment/Comment/Base.lproj/Main.storyboard
ソース	Bonus/Chapter3/Comment/Comment/ViewController.swift

Xcodeで使える特殊なコメント表記には、以下のようなものがあります。

表記	意味
MARK	プログラムの区切りなどに印をつける
TODO	積み残しの課題を記載する
FIXME	修正箇所を明示する

カウンターアプリの`ViewController.swift`を開きます。

まず、カウント用の変数が定義されていることが分かります。

サンプルプログラム　Bonus/Chapter3/Comment/Comment/ViewController.swift

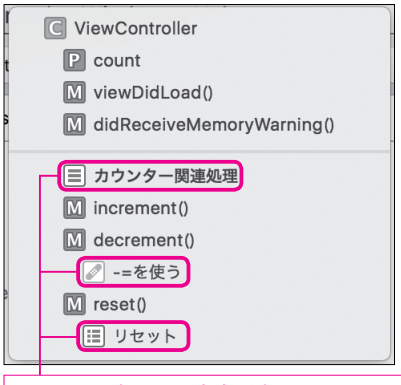
```
var count = 0
```

続きを読み進めていくと、次の部分で特殊なコメントを使っていることが分かります。

サンプルプログラム　Bonus/Chapter3/Comment/Comment/ViewController.swift

```
// MARK: - カウンター関連処理

func increment() {
    count += 1
}

func decrement() {
    // FIXME: -=を使う
    count = count - 1
}

func reset() {
    // TODO: リセット
}
```

このように書いておけば、Xcodeが特殊なコメントを認識して表示してくれます。



MARK: -のように -をつけることで区切り線が入ります。適切にコメントを入れておくと、後から編集しやすくなりますので、利用してみてください。

Section 3.6

自動テスト

アプリをテストする一番単純な方法は、手動・目視での確認です。しかし、この方法は手間がかかるという難点があります。

例えば、アルファベットの小文字で入力した文字列を、大文字に変換するアプリがあると仮定します。

最初は単純に大文字に変換するだけでしたが、文字列が未入力の場合にはエラーメッセージを表示したくなりました。すると、以下の両方を確認しなくてはなりません。

- **元々実装していた大文字に変換する処理**
- **新たに実装したエラー表示の処理**

このような時に自動テストが役に立ちます。大文字変換の機能を確認する自動テストが既にあれば、それを実行するだけで元々あった仕様の確認は完了です。あとはエラーメッセージ表示用の自動テストも追加するか、そこだけ目視確認するなどで確認完了です。

自動テストを書く

例にあげた大文字返還アプリをサンプルとして、テストを書いてみましょう。

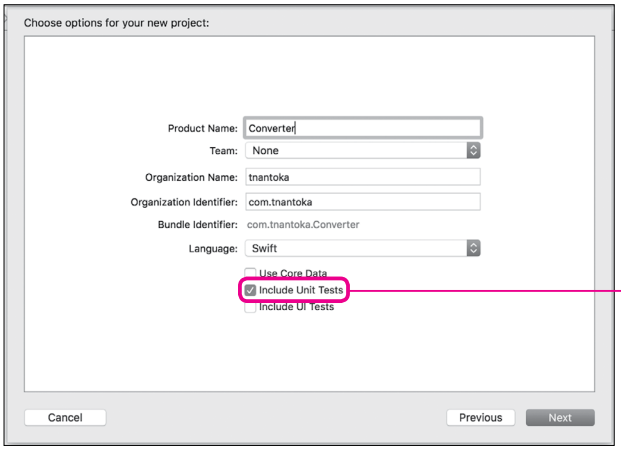
サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter3/Converter.xcodeproj
Storyboard	Bonus/Chapter3/Converter/Converter/Base.lproj/Main.storyboard
ソース	Bonus/Chapter3/Converter/Converter/ViewController.swift

1. アプリを用意する

入力した文字を大文字に変換する簡単なアプリです。

プ新規Xcodeプロジェクトのオプション選択画面で「Include Unit Tests」をオンにすることで、ロジェクト作成時にテストを一緒に作成することができます。



テキストフィールド、ラベルをOutletで用意しておきます。

サンプルプログラム　Bonus/Chapter3/Converter/Converter/ViewController.swift

```
@IBOutlet weak var textField: UITextField!
@IBOutlet weak var label: UILabel!
```

ボタンから呼ばれるonTapアクションは以下のような単純な処理です。

サンプルプログラム　Bonus/Chapter3/Converter/Converter/ViewController.swift

```
if let text = textField.text {
    label.text = text.uppercased()
}
```

2. テストを書く

ConverterTests.swiftファイルにテストを追加します。Buttonのアクションに設定されたonTapメソッドが呼ばれた時に、テキストが大文字に変換されているかを確認しています。

サンプルプログラム　Bonus/Chapter3/Converter/ConverterTests/ConverterTests.swift

```
func testConvert() {
    // Storyboardファイルからコントローラーを取得
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let controller = storyboard.instantiateInitialViewController() as!
ViewController

    // ビューにアクセスすることで、ビューを読み込む
    _ = controller.view

    // 変換テスト
    // テキストを設定
    controller.textField.text = "text"
    // ボタンのタップと同じことをする
    controller.onTap(self)
    // テキストが"TEXT"になっているか確認
    XCTAssertEqual(controller.label.text!, "TEXT")
}
```

本来このようなロジック部分はモデルファイルなどを作って、コントローラーとは別ファイルに記載すべきと思われますが、ここでは簡略化してコントローラー内に実装してそのままテストします。

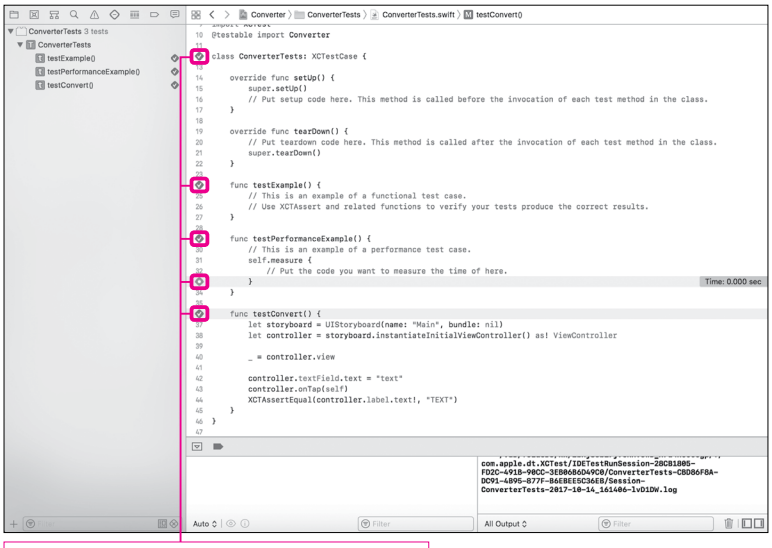
なお、テストでは予期せぬnilが入ってきた場合などに早期にクラッシュする方が望ましいことも多いので、!を気にせず使います。

Note

テストファイルを分けたい場合は「Unit Test Case」を追加します。
「UI Test Case」と間違えないように注意してください。
UIの方はシミュレーターを自動操作して行なうテストです。

3. テストを実行する

ショートカット command＋ Uでテストを実行することができます。緑のチェックマークが成功という意味です。



緑のチェックマークが表示されれば成功です。

4. 仕様変更

テキストが未入力の場合、エラーを表示する仕様変更を行います。

サンプルプログラム　Bonus/Chapter3/Converter/Converter/ViewController.swift

```
if let text = textField.text {
    if text.isEmpty {
        label.text = "テキストを入力してください"
    } else {
        label.text = text.uppercased()
    }
}
```

この状態で再度テストを実行（command＋U）すると問題なく通過しました。仕様の変更によって、既存の処理がおかしくなってないことはこれで確認でき一安心です。後はエラーメッセージの処理を確認すれば仕様変更は完了です。

リファクタリングとテスト

アプリの仕様変更以外にも、リファクタリングや、Swiftのバージョンアップへの対応などで、プログラムの変更が必要となります。

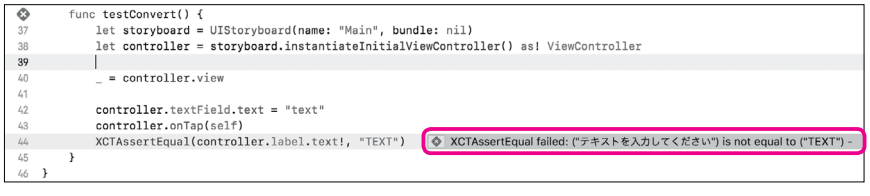
筆者は自動テストの一番のありがたみはリファクタリングがしやすくなることだと考えています。プログラムをより気軽に直せるというのは、精神衛生上とてもいいものです。

例えば、先程の処理をguardで書き換えました。

サンプルプログラム　Bonus/Chapter3/Converter/Converter/ViewController.swift

```
guard let text = textField.text else { return }
guard text.isEmpty else {
    label.text = "テキストを入力してください"
    return
}
label.text = text.uppercased()
```

単純な変更であり、テストがないと確認が面倒で飛ばしてしまいそうになります。しかし、実は条件が逆になってしまっています。テストを実行するとすぐ気づきます。



正しくは次の通りです。

サンプルプログラム　Bonus/Chapter3/Converter/Converter/ViewController.swift

```
guard let text = textField.text else { return }
guard !text.isEmpty else {
    label.text = "テキストを入力してください"
    return
}
label.text = text.uppercased()
```

これならテストも通りました。

```
    func testConvert() {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let controller = storyboard.instantiateInitialViewController() as! ViewController

        _ = controller.view

        controller.textField.text = "text"
        controller.onTap(self)
        XCTAssertEqual(controller.label.text!, "TEXT")
    }
}
```

全ての機能に対してテストを書くのはとても大変ですが、大事な処理だけでも書いてあると安心です。

Note

どこまでテストを書くのか、というのは難しい問題です。

特にiOSは1年に1回大幅なアップデートが行われることもあり、書いたテストが無駄になってしまう可能性も考えられます。

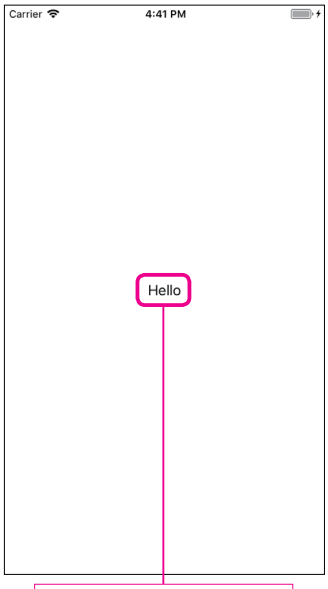
筆者は「このアプリにとってこの機能がなければ絶対困る」という部分は少なくともテストを書くべきだと考えています。

例えばメモ帳アプリであれば「メモを作成・変更して保存、それを表示できる」というのはなくてはならないものです。一方フォントの変更ができるという機能は、必須ではないと考えることもできるでしょう。

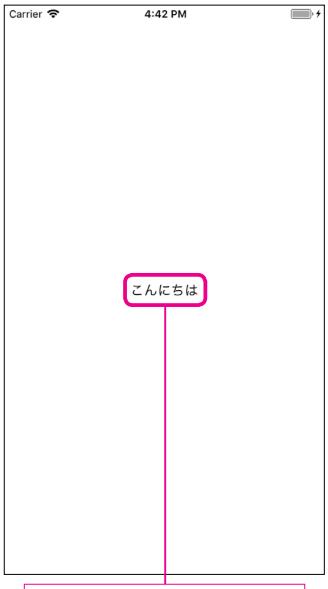
Section

3.7 多言語対応

サンプルアプリ



英語で実行した場合の表示。



日本語で実行した場合の表示。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter3/Localization.xcodeproj
Storyboard	Bonus/Chapter3/Localization/Localization/Base.lproj/Main.storyboard
ソース	Bonus/Chapter3/Localization/Localization/ViewController.swift

多言語対応とは

アプリの利用者は日本に住んでいる人より、英語圏に住んでいる人の方が圧倒的に多いです。絶対に日本語でしか使われないというアプリでない限り、英語対応しておくことをお勧めします。そうすることで沢山の人に使ってもらえる可能性が広がります。

多言語対応はローカリゼーションや国際化（インターナショナリゼーション・i18n）とも呼ばれます。Xcodeでの多言語対応はとても簡単です。

今まではソースコードに直接日本語で文字列を書いてきましたが、多言語化する場合は、英語を基準として日本語化するイメージになります。また、Storyboard上でも対応可能ですが、保守が大変なのでコード上で指定することをおすすめします。

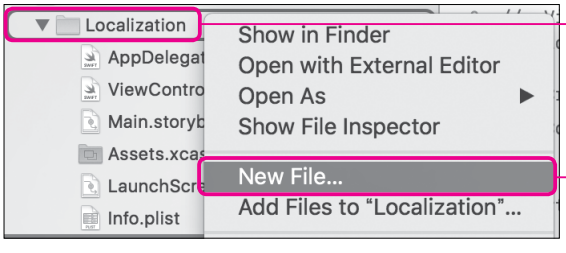
事前準備：アプリを用意する

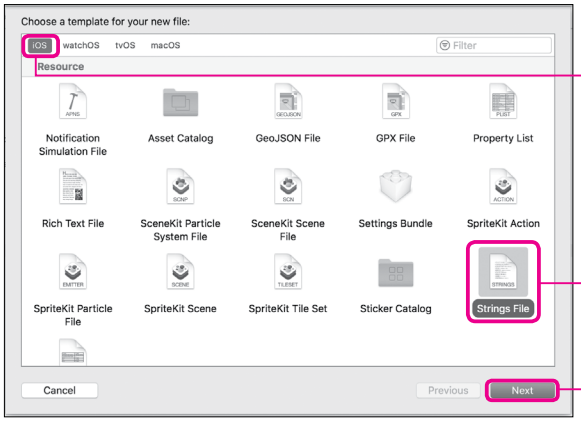
ここでは、例としてラベルのテキストに英語では「Hello」と、日本語では「こんにちは」と表示させてみます。

Storyboardの上下左右中央にラベルを配置し、「Assistant Editor」で「label」として Outlet接続しておきます。また、ラベルのテキストと「Hello」と書き換えておきます。

1. Localizable.stringsの作成

言語情報を管理するファイルを作ります。

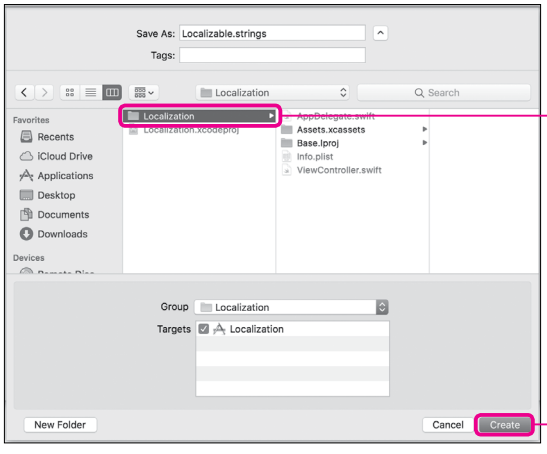




①「iOS」を選択します。

②「Strings File」を選択します。

③「Next」をクリックします。

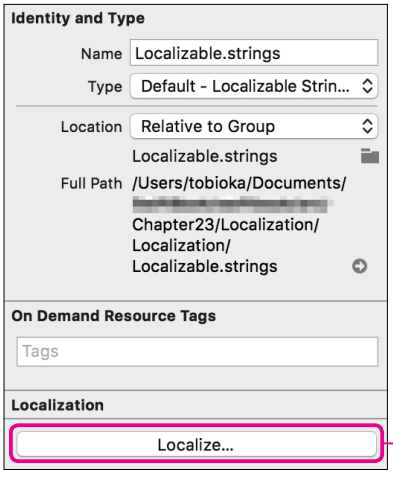


①「Localizable.strings」という名前を付けて作成します。

②「Create」をクリックします。

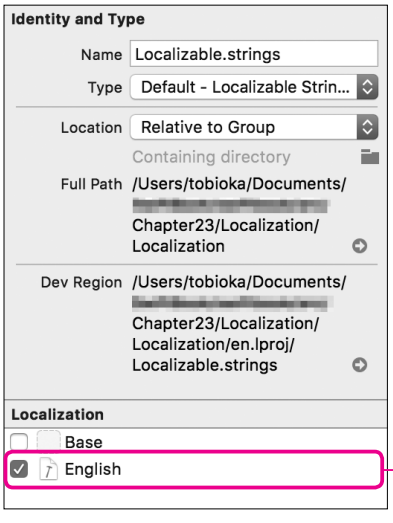
2. Localizable.stringsの多言語化

作成されたファイルは多言語化の対象になっていないため、「File inspector」からLocalizeします。



①「Navigator Area」で「Localizable.strings」を選択した状態で「File inspector」を開きます。

②「Localize...」をクリックします。



「English」にチェックが入っています

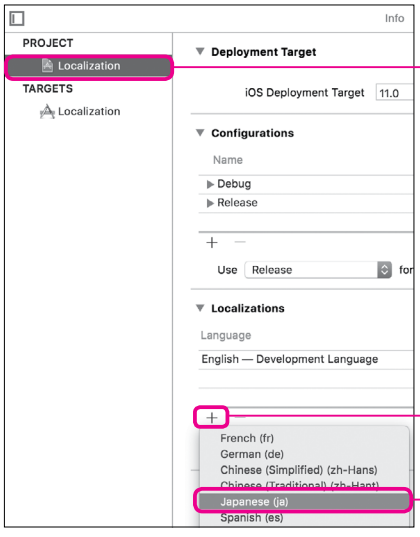
英語用の情報として以下のように記述します。「Hello」を「Hello」に翻訳するという意味になります。(英語なのでそのままです。)

サンプルプログラム　Bonus/Chapter3/Localization/Localization/en.lproj/Localizable.strings

```
"Hello" = "Hello";
```

3. 日本語の追加

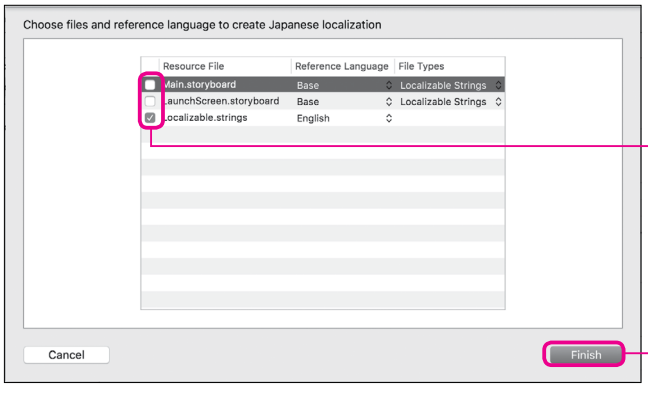
言語として日本語を追加します。



①「Navigator Area」で最初の項目を選び、「Document Outline」で「PROJECT」の「Localization」を選択します。

②「＋」をクリックします。

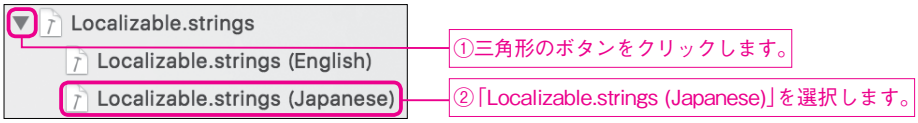
③「Japanese (ja)」を選択します。

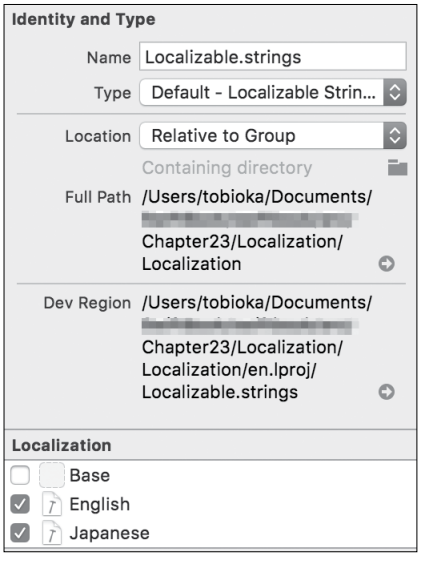


①「Localizable.strings」だけをオンにして、他をオフにします。

②「Finish」をクリックします。

「Navigator Area」を見ると日本語用の`Localizable.strings`が作成されていることを確認できるので、これを編集して日本語の言語情報を記載します。





「Hello」が「こんにちは」に翻訳されるように記述します。

サンプルプログラム　Bonus/Chapter3/Localization/Localization/ja.lproj/Localizable.strings

```
"Hello" = "こんにちは";
```

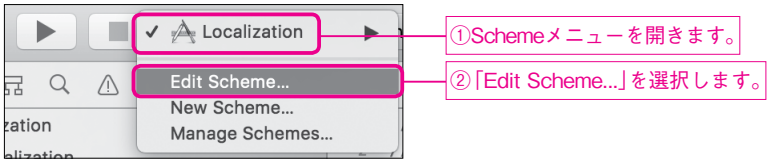
4. 多言語化したテキストの表示

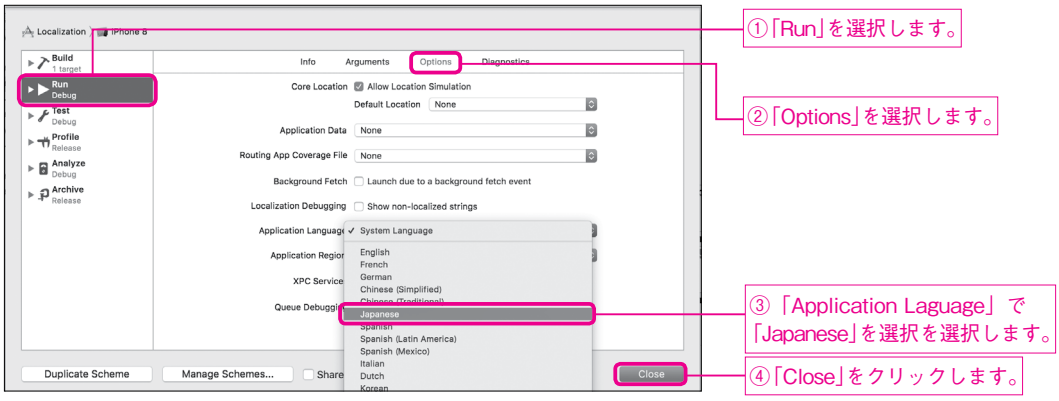
NSLocalizedStringを使って、多言語化した"Hello"を呼び出します。commentには空文字列を指定しておけばOKです。

サンプルプログラム　Bonus/Chapter3/Localization/Localization/ViewController.swift

```
label.text = NSLocalizedString("Hello", comment: "")
```

言語を切り替えて多言語化されたテキストの表示を確認します。





実行すると「こんにちは」と表示されます。

Note

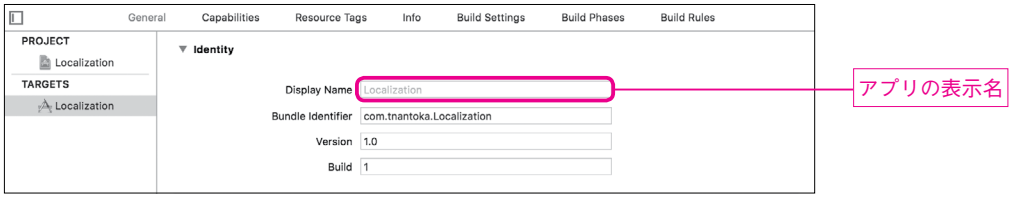
この手順で毎回設定を変えるのは大変です。

基本は日本語に設定にしておいて、英語を確認したいときはシミュレーターのホーム画面からアプリを起動すると便利です。

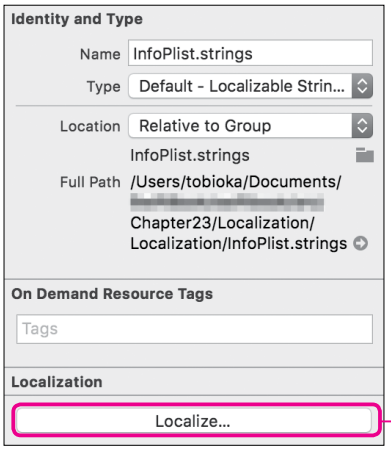
Xcode経由でアプリを実行しなければ、Schemeメニューで行った設定は反映されないため、ホーム画面から起動した場合はシミュレーターの言語（デフォルト設定だと英語）が使われます。

アプリ名

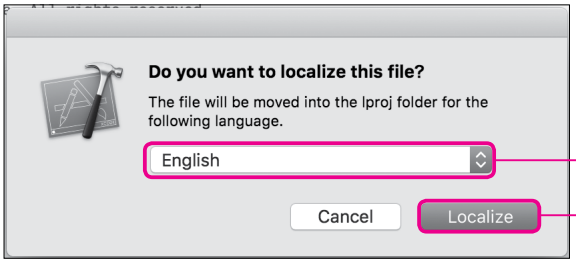
アプリの表示名は、「Identity」の「Display Name」から変更可能です。



しかし、言語ごとに表示されるアプリ名を変更したい場合はローカライズが必要です。`Localizable.strings`と同じように`InfoPlist.strings`を作成し、以下の手順で多言語化します。

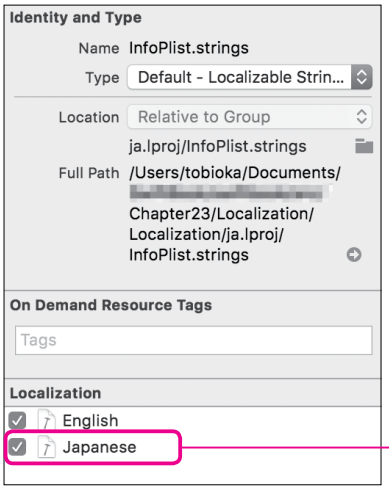


①「Localize」をクリックします。



②「English」を選択します。

③「Localize」をクリックします。



④「Japanese」をオンにします。

英語のアプリ名を設定します。

サンプルプログラム　Bonus/Chapter3/Localization/Localization/en.lproj/InfoPlist.strings

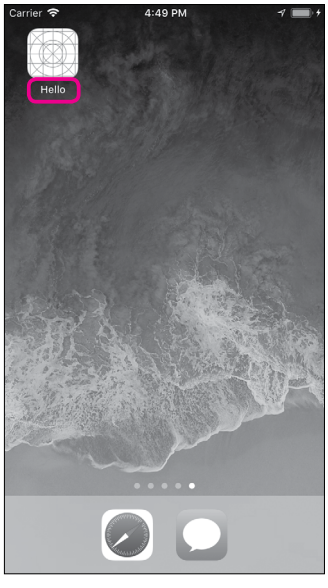
```
CFBundleDisplayName = "Hello";
```

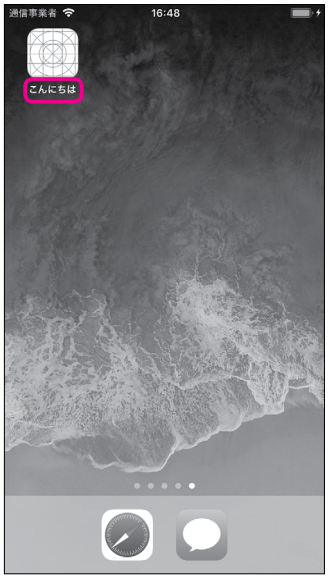
日本語のアプリ名も設定します。

サンプルプログラム　Bonus/Chapter3/Localization/Localization/ja.lproj/InfoPlist.strings

```
CFBundleDisplayName = "こんにちは";
```

実機のiPhoneと同じ手順で、シミュレーターの「設定（Setteings）」アプリから「一般」→「言語と地域」→「iPhoneの使用言語」を日本語に変えると、アプリ名が日本語になります。（Schemeメニューから変える方法だと反映されません。）





ここではXcodeの様々な使い方を紹介しました。
気になる機能があればぜひ開発に取り入れてみてください。

『Swift 4プログラミング入門』ボーナスPDF

Chapter 4

外部ライブラリを使う

本格的なアプリを作ろうとすると、すべての機能を1から作るのは現実的でなくなってきます。幸いなことに現在はたくさんのライブラリがオープンソースソフトウェアとして公開されています。それらのライブラリを有効活用させてもらい、開発を効率化しましょう。

Section 4.1

ライブラリ依存管理ツール

ライブラリのソースコードをコピーしてきてプロジェクトに取り込めば、他の人が書いたコードを使うことはできます。しかし、その方法だとバージョンアップやライブラリの依存関係の解決などが大変になります。依存管理ツールを使えば、バージョンや依存関係を管理してくれます。

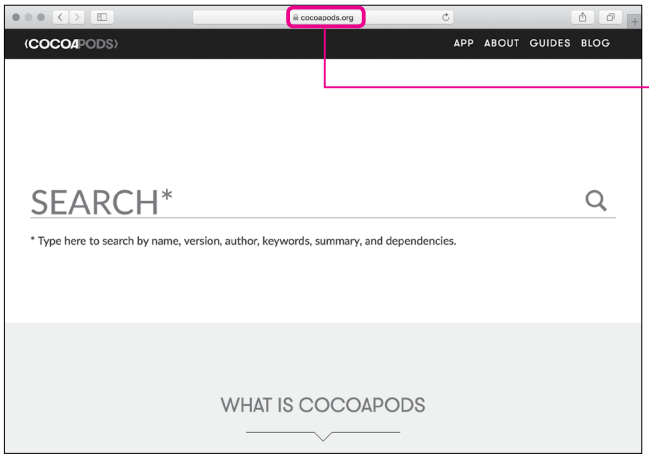
Note

依存関係とは、例えば「ライブラリ Aを使うためにはライブラリ Bが必要で、さらにライブラリ Bを使うためにはライブラリCが必要」というような関係です。これを手動で管理するのは大変です。

Swiftに使える代表的な依存管理ツールは以下の3つです。

- **CocoaPods**
- **Swift Package Manager**
- **Carthage**

CocoaPods



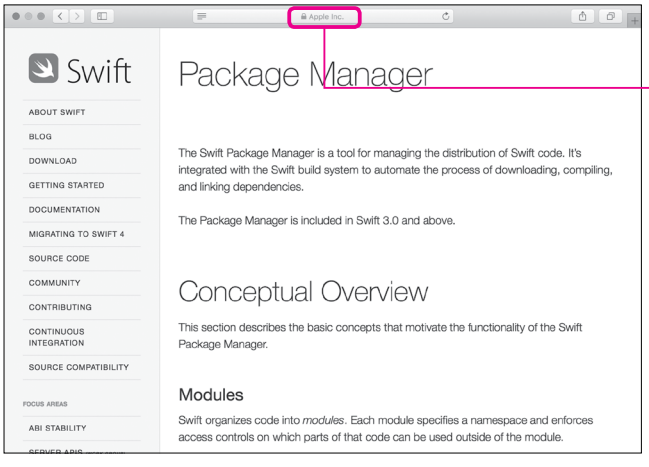
「CocoaPods」
https://cocoapods.org

CocoaPodsは Objective-C（Swift登場以前から iOS開発に使われてきた言語）の時代から使われている定番のライブラリ依存管理ツールです。簡単なコマンドを実行するだけで、依存管理に関する様々な仕事を簡略化してくれる高機能なツールです。CocoaPodsに対応しているライブラリの情報が一元管理されており、公式サイトで検索できることも魅力です。

一方、様々なことを自動で行なうため、CocoaPodsが原因でエラーになった場合、解決に手間取ることがあります。また、Rubyというスクリプト言語で作られているので、Rubyの経験がないと戸惑う可能性もあります。

筆者は数年間お世話になっていますが、予期せぬエラーが発生した時のサポートが書籍では難しいこともあるため、今回は採用しないことにします。

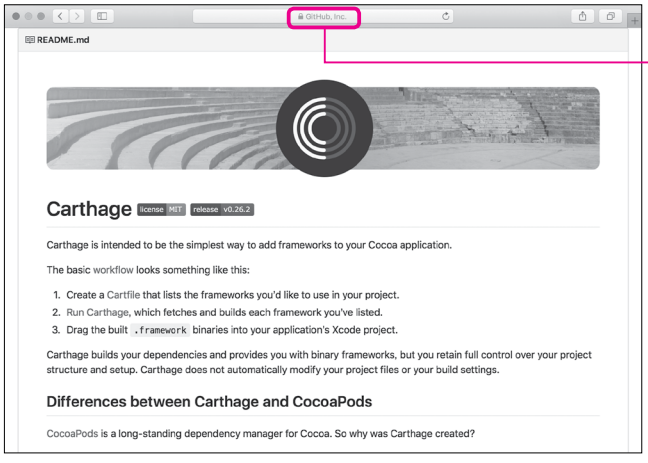
SPM（Swift Package Manager）



「SPM（Swift Package Manager）」
https://swift.org/package-manager/

SPMは Apple公式のツールです。現在はコマンドラインツールや Linuxで使われていますが、iOSでもいずれこちらが標準になっていくことでしょう。現状ではまだ iOSに使う環境が整っていないため、本書では使用しません。

Carthage



「Carthage」
https://github.com/Carthage/Carthage#readme

Carthageは、GitHub社が 2014年に公開した、比較的新しいライブラリ依存管理ツールです。Swiftで作られており、pkgファイルも提供されているのでインストールが簡単です。

CocoaPodsと違い、最低限の管理しか行いません。そのため手作業での設定が多くなりますが、Carthageが原因で問題が起こる可能性が低くなっています。また、CocoaPodsよりもビルドの時間が短く済むという特徴もあるため、使用するライブラリが増えてくるとその点で優位です。

インストールの簡単さや問題が起こりにくい点を重視して、本書ではCarthageを使用します。

Note

Carthageは英語では「カッセージ」のような読み方になりますが、日本では「カルタゴ」と読まれることもあります。

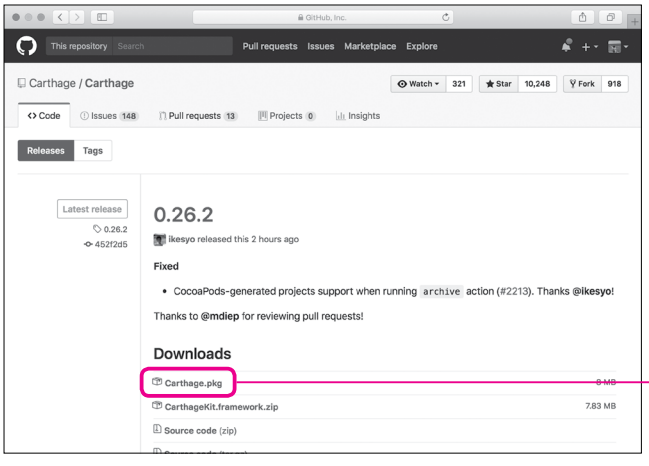
Section 4.2

Carthageの使い方

Carthageを使うには事前にインストールが必要です。

ダウンロード

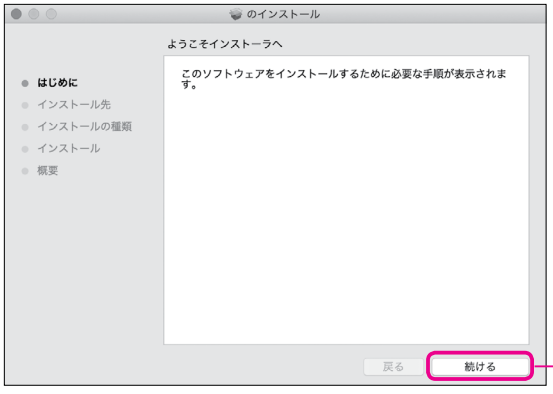
Carthageは次のサイトから入手することができます。
https://github.com/Carthage/Carthage/releases



ダウンロードしたいバージョンのCarthage.pkgをダウンロードします。ここでは0.26.2を使います。

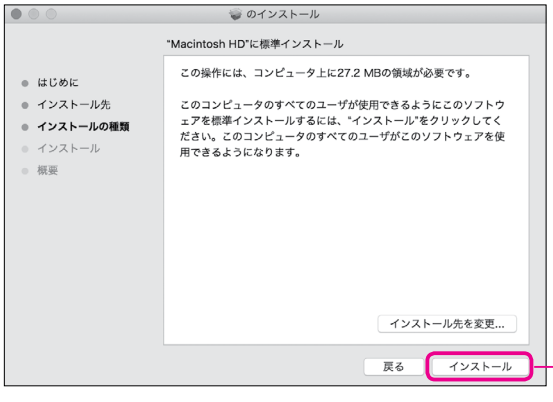
インストール

ダウンロードしたpkgファイルをダブルクリックで開きインストールします。
インストーラーに従って進めます。通常のアプリと一緒で特に変わった手順は必要ありません。

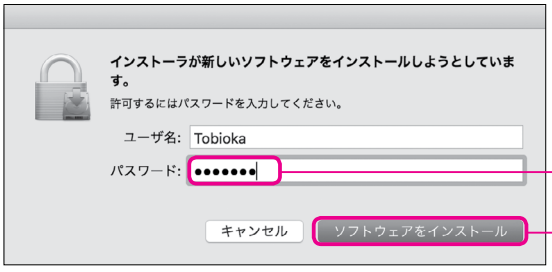


「続ける」をクリックします。

「続ける」をクリックします。

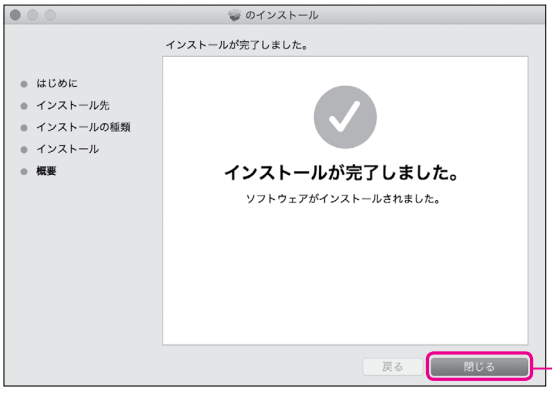


「インストール」をクリックします。

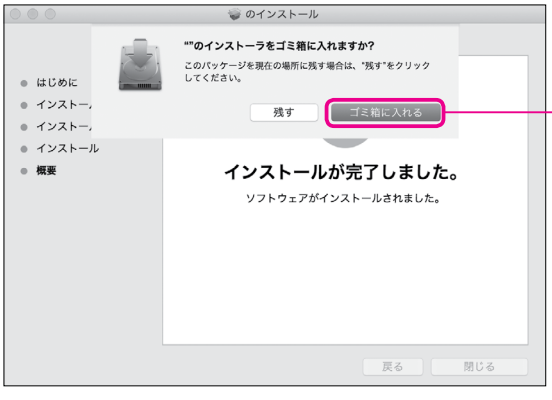


①「パスワード」を入力します。

②「ソフトウェアをインストール」をクリックします。



「閉じる」をクリックします。



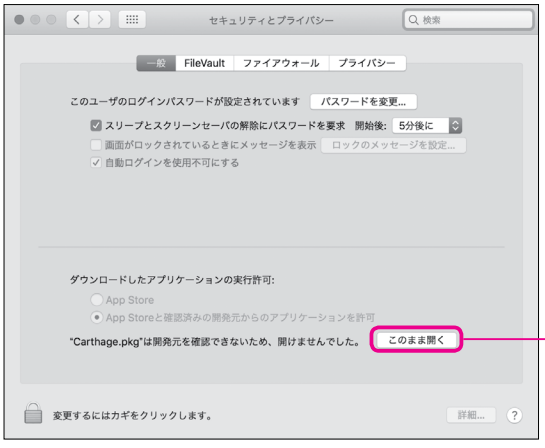
「ゴミ箱に入れる」をクリックします。

セキュリティ設定

「開発元が未確認」という警告が表示される場合は、セキュリティ設定から「このまま開く」を選択することでインストールできます。

「OK」をクリックして警告画面を閉じます。

「セキュリティとプライバシー」をクリックします。



「このまま開く」をクリックします。

「開く」をクリックします。

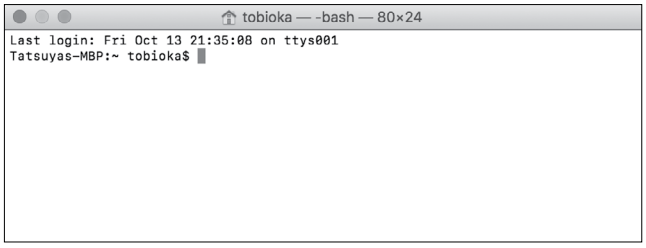
Note

2018年 3月時点では Carthageが悪意のあるプログラムでないことを確認しています。ただし、この設定はセキュリティを一時的に弱めることになるため、開くプログラムが間違っていないかなどを確認し、慎重に行なってください。

バージョンの確認

依存管理ツールはコマンドラインから使用します。苦手意識がある方も、必要なコマンドは数コマンドなので、本書を見ながら実行してみてください。

アプリケーションフォルダのユーティリティの中にあるターミナル.appを起動します。



以下のコマンドを入力してエンターキーを押します。先頭の$記号はターミナルでの入力を表す記号で実際に入力するのはその後に続く内容のみです。↵はreturn（Enter）キーを示すものです。

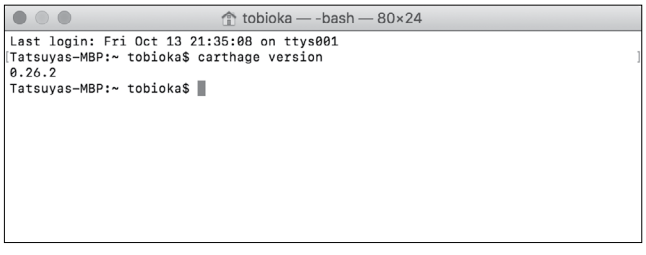
ターミナル

```
$ carthage version↵
```

実行結果

```
0.26.2
```

先程ダウンロードしたバージョンが表示されていればインストールは成功です。



ライブラリの管理方法

Carthageによるライブラリ管理は以下の流れで行います。少し複雑ですが、ライブラリ追加のたびに毎回行なう作業なのでいずれ慣れるでしょう。

1. **Cartfileの作成**
2. **ビルド**
3. **Linked Frameworks and Librariesの追加**
4. **Run Scriptの設定**

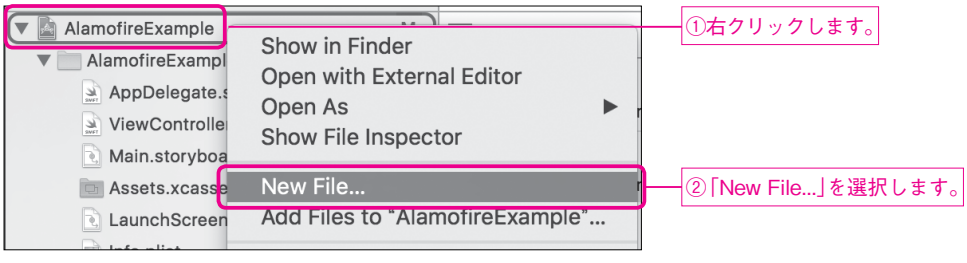
5. **Input Filesの設定**
6. **Output Filesの設定**
7. **ライブラリを使用する**

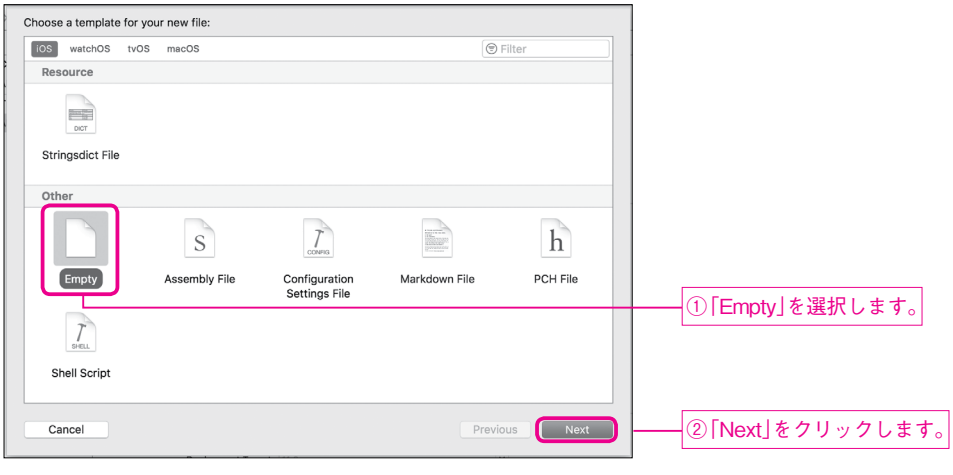
サンプルファイル

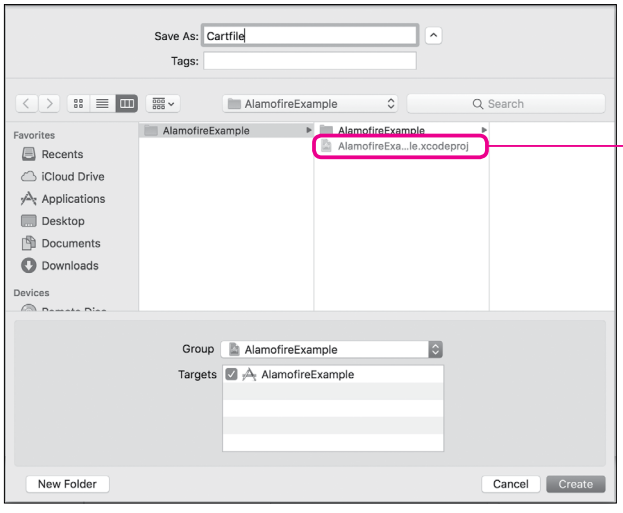
種類	ファイル
プロジェクト	Bonus/Chapter4/AlamofireExample.xcodeproj
Storyboard	変更しません
ソース	Bonus/Chapter4/AlamofireExample/AlamofireExample/ViewController.swift

1. Cartfileの作成

`Cartfile`というファイルに使用したいライブラリを記載します。このファイルはプロジェクトフォルダー（プロジェクト名`.xcodeproj`ファイルがあるフォルダー）に作ります。







例えば、Alamofire（https://github.com/Alamofire/Alamofire）というライブラリのバージョン4.5.1を使いたい場合は以下のように記載します。

サンプルプログラム　Bonus/Chapter4/AlamofireExample/Cartfile

```
github "Alamofire/Alamofire" == 4.5.1
```

複数行書くことで複数のライブラリを使用できます。

Note

Alamofireは最も有名な Swift製のネットワーク通信ライブラリです。Chapter 20では URLSessionを用いた通信方法を紹介しましたが、複雑な通信をする際はAlamofireの利用を検討してみてもよいでしょう。

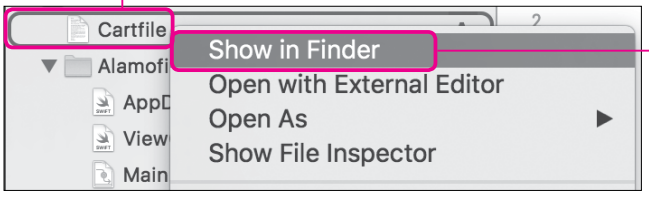
2. ビルド

以下のコマンドを叩いてライブラリをダウンロード及びビルドします。

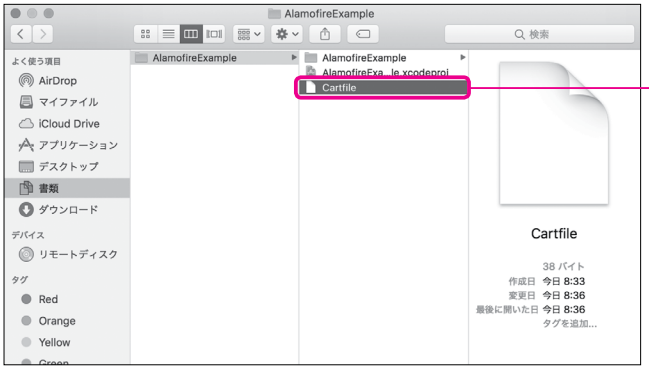
`cd`はチェンジディレクトリーの意味でプロジェクトフォルダーまで移動するコマンドです。`cd`（c、d、半角スペース）まで入力した後、ターミナルにフォルダーをドラッグ&ドロップすれば場所が入力されるので、そのままreturnキーを押せばOKです。

（`/Users/`ユーザー名`/`フォルダー名`/`プロジェクト名のような形式になります。）

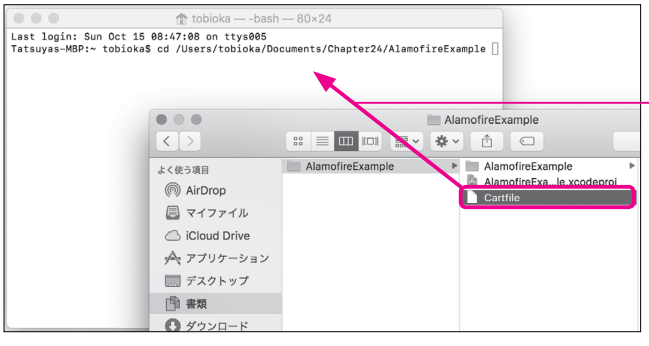
①「Cartfile」を右クリックします。



②「Show in Finder」を選択します。



Finderでフォルダーが開きます。



「cd 」まで入力したターミナルにフォルダーをドラッグ＆ドロップします。

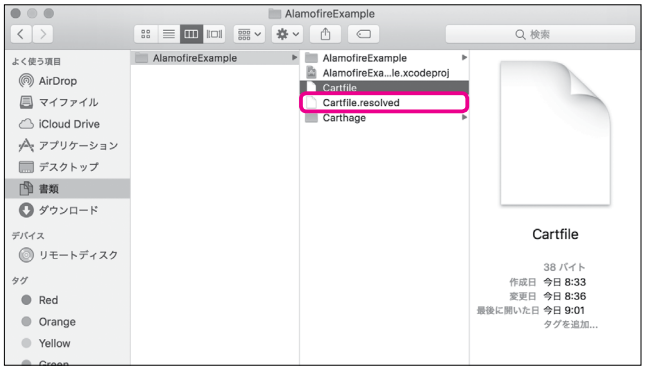
ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

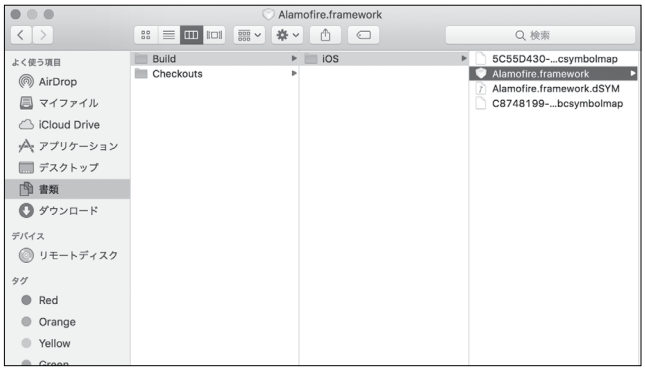
```
*** Fetching Alamofire
*** Checking out Alamofire at "4.5.1"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.2B4LYo.log
*** Building scheme "Alamofire iOS" in Alamofire.xcworkspace
```

errorなどのメッセージが出ておらず、Cartfile.resolvedというファイルと Carthageというフォルダーが作られていれば成功しています。

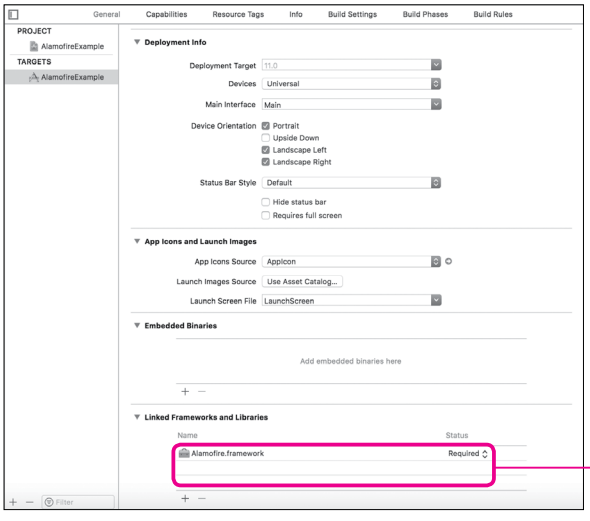


3. Linked Frameworks and Librariesの追加

Carthageフォルダーの中をBuild、iOSとたどるとライブラリ名.frameworkというファイルが見つかります。今回であればAlamofire.frameworkです。



そのAlamofire.frameworkをXcodeの「Linked Frameworks and Libraries」にドラッグ&ドロップして追加します。



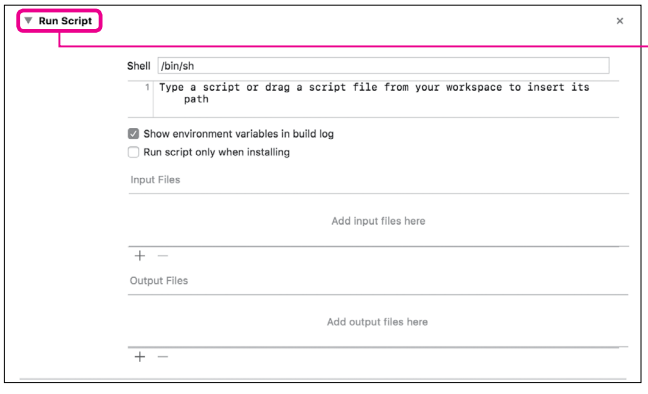
最下部の「Linked Frameworks and Libraries」にドラッグ&ドロップします。

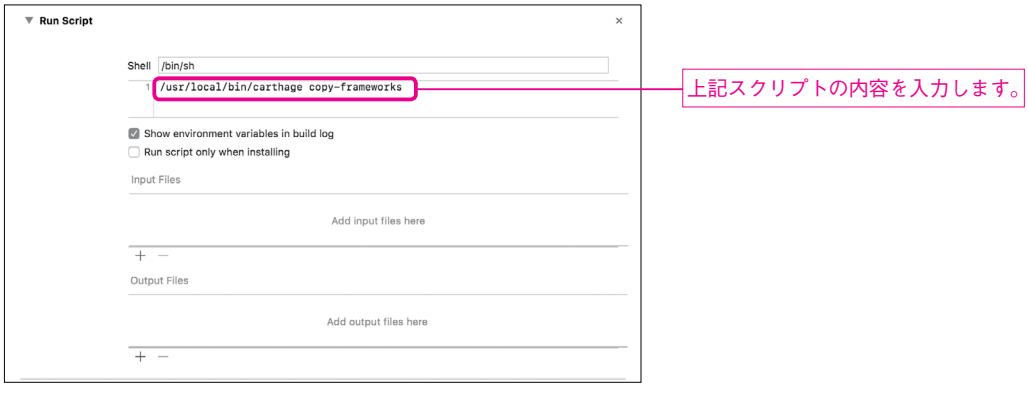
4. Run Scriptの設定

「Build Phases」を開き、左上の「+」から「New Run Script Phase」を選択します。

追加された「Run Script」の左側の三角形をクリックし、「Shell」の右下のテキストボックスに以下の内容を入力します。

```
/usr/local/bin/carthage copy-frameworks
```

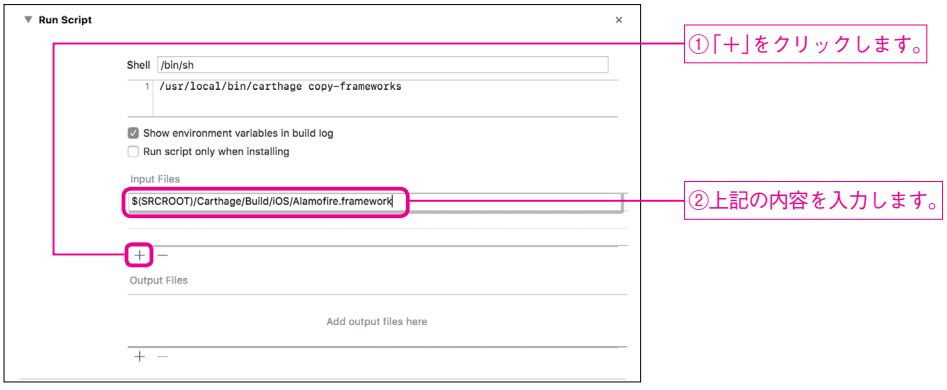




5. Input Filesの設定

スクリプトの下の「Input Files」の「+」をクリックし、以下の内容を入力します。

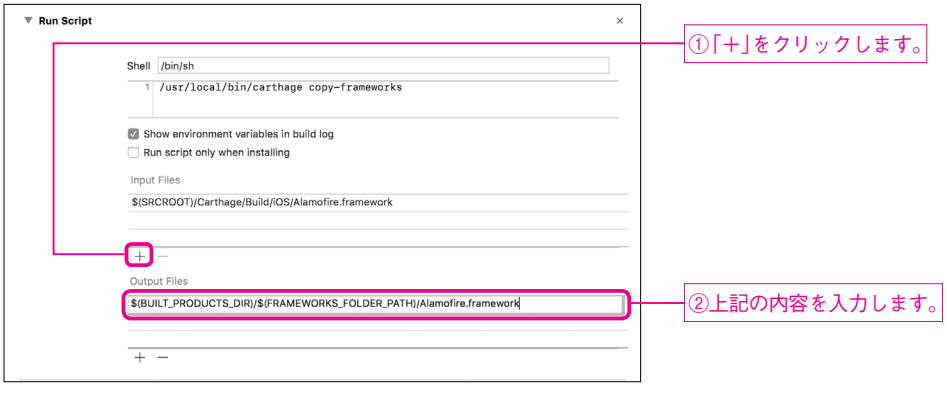
```
$(SRCROOT)/Carthage/Build/iOS/Alamofire.framework
```



6. Output Filesの設定

「Output Files」にも同様にして、以下の内容を追加します。

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/Alamofire.framework
```



7. ライブラリを使用する

これで準備は整いました。Alamofireはネットワークのライブラリなので、例として `http://www.socym.co.jp/` のコンテンツを取得してみましょう。

ライブラリを`import`します。

サンプルプログラム　Bonus/Chapter4/AlamofireExample/AlamofireExample/ViewController.swift

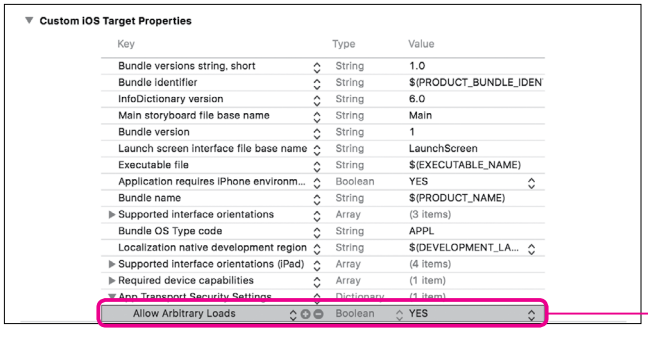
```
import Alamofire
```

viewDidLoadで通信します。

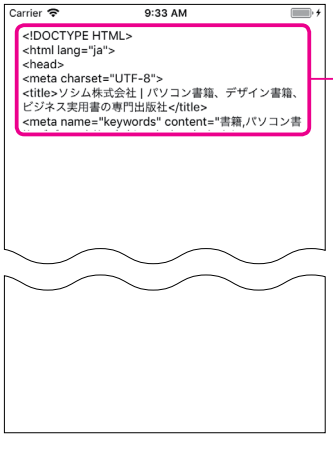
サンプルプログラム　Bonus/Chapter4/AlamofireExample/AlamofireExample/ViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()

    Alamofire.request("http://www.socym.co.jp/").responseString {
        response in self.textView.text = response.result.value
    }
}
```

HTTPS通信ではないのでATSで許可します。



「Allow Arbitrary Loads」をYesにします。



配置しておいたテキストビューに内容が表示されています。

このような手順で外部ライブラリを利用したアプリを開発することができます。他のライブラリでも基本的に手順は同じです。

本 PDFの Section 3.3以降では、定番のライブラリをいくつか紹介します。よく使われているライブラリは品質が担保され、安心感があります。選定の参考にしてください。Objective-Cで開発されたライブラリもたくさんありますが、本書ではSwiftで開発されたライブラリを紹介します。

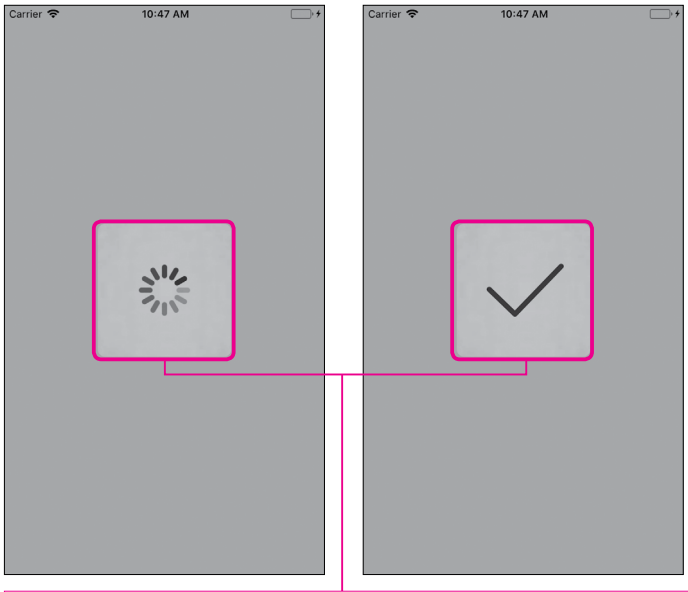
Note

利用するライブラリを増やし過ぎると、プロジェクトの維持・管理が煩雑になりますので注意してください。

Section 4.3

ローディング—PKHUD

サンプルアプリ



アプリを起動すると一定時間ローディングが表示され、成功表示の後消えます。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/PKHUDExample.xcodeproj
Storyboard	変更しません
ソース	Bonus/Chapter4/PKHUDExample/PKHUDExample/ViewController.swift

PKHUDについて

通信中や時間のかかるデータ処理など、やむを得ずアプリのユーザーを待たせるケースがあります。その際にはローディング表示をしてあげると親切です。

単純なインジケーターの表示なら、UIActivityIndicatorViewを使って表示することもできます。しかし、ローディング表示を最前面に出したり、画面をロック（操作不可に）したりするのは、意外に手間がかかります。

PKHUD（https://github.com/pkluz/PKHUD）を使うとローディング表示を簡単に実現することができます。

サンプルアプリの作成

1. Cartfileの作成

Cartfileに以下を追記します。

サンプルプログラム　Bonus/Chapter4/PKHUDExample/Cartfile

```
github "pkluz/PKHUD" == 5.0.0
```

2. ビルド

`update`コマンドを実行し、ビルドします。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

```
*** Fetching PKHUD
*** Checking out PKHUD at "5.0.0"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.2b0GT7.log
*** Building scheme "PKHUD" in PKHUD.xcodeproj
```

3. Linked Frameworks and Librariesの追加

Carthage/Build/iOS/PKHUD.frameworkを「Linked Frameworks and Libraries」にドラッグ&ドロップします。

4. Run Script

Run ScriptもAlamofireの時と同様に追加します。

スクリプト

```
/usr/local/bin/carthage copy-frameworks
```

Input Files

```
$(SRCROOT)/Carthage/Build/iOS/PKHUD.framework
```

Output Files

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/PKHUD.framework
```

5. HUDの表示

準備が整ったのでPKHUDを使用してHUDを表示してみます。ライブラリをimportします。

サンプルプログラム　Bonus/Chapter4/PKHUDExample/PKHUDExample/ViewController.swift

```
import PKHUD
```

viewDidAppearで表示します。ローディング表示し3秒後に消した後、成功の表示を1秒間出しています。

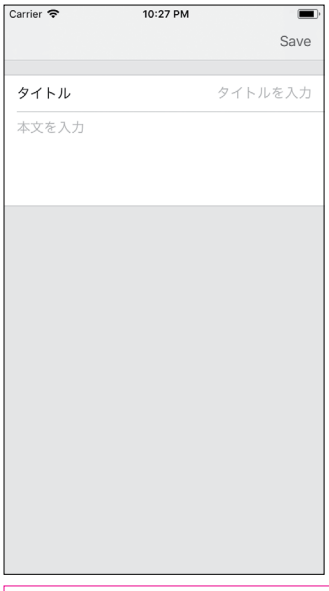
サンプルプログラム　Bonus/Chapter4/PKHUDExample/PKHUDExample/ViewController.swift

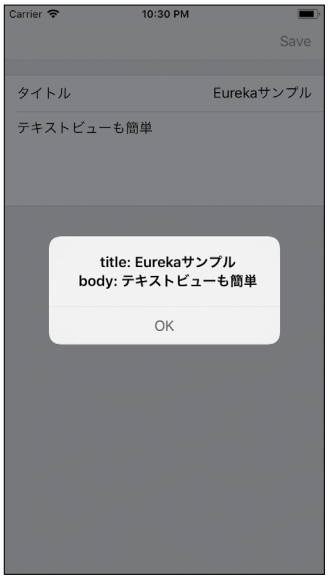
```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    HUD.show(.progress)                    // 読み込み中の表示
    HUD.hide(afterDelay: 3.0) { _ in       // 3秒後に今の表示を消す
        HUD.flash(.success, delay: 1.0) // 成功を表示して1秒後に消す
    }
}
```

Section 4.4

フォーム作成—Eureka

サンプルアプリ





テキスト入力フォームを表示します。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/EurekaExample.xcodeproj
Storyboard	変更しません
ソース	Bonus/Chapter4/EurekaExample/EurekaExample/ViewController.swift

Eurekaについて

ユーザーに何か入力させたい場合にはフォームが必要です。HTMLではとても簡単なこの作業ですが、iOSアプリでは少し大変です。

タイトルと本文を入力するフォームを考えてみましょう。

フォームらしい見た目にするには TableViewを使う必要があります。そして各セルに TextFieldやTextViewを配置して、Delegateなどを適切に処理する必要があります。これはとても骨の折れる作業です。フォーム機能がアプリのメイン機能ではない場合は特にライブラリに頼るのが得策です。

Eureka（https://github.com/xmartlabs/eureka）を使えば iOS標準アプリのようなフォームが簡単に作れます。

1 サンプルファイルの作成

1. Cartfileの作成

Cartfileに次の内容を記載します。

サンプルプログラム　Bonus/Chapter4/EurekaExample/Cartfile

```
github "xmartlabs/Eureka" == 4.0.1
```

2. ビルド

次のようにビルドします。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

```
*** Cloning Eureka
*** Checking out Eureka at "4.0.1"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.m7VFbn.log
*** Building scheme "Eureka" in Eureka.xcworkspace
```

3. Linked Frameworks and Librariesの追加

Carthage/Build/iOS/Eureka.frameworkを Linked Frameworks and Librariesにドラッグ＆ドロップします。

4. Run Script

次のように設定します。

スクリプト

```
/usr/local/bin/carthage copy-frameworks
```

Input Files

```
$(SRCROOT)/Carthage/Build/iOS/Eureka.framework
```

Output Files

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/Eureka.framework
```

5. フォームの表示

タイトルと本文が入力できるフォームを表示します。

Eurekaを`import`します。

サンプルプログラム　Bonus/Chapter4/EurekaExample/EurekaExample/ViewController.swift

```
import Eureka
```

FormViewControllerを継承します。

サンプルプログラム　Bonus/Chapter4/EurekaExample/EurekaExample/ViewController.swift

```
class ViewController: FormViewController {
```

フォームの設定を行います。

サンプルプログラム　Bonus/Chapter4/EurekaExample/EurekaExample/ViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()

    // タイトルを入力するRow（1行）
    let titleRow = TextRow { row in
        row.title = "タイトル"
        row.placeholder = "タイトルを入力"
    }

    // 本文を入力するRow（複数行）
    let bodyRow = TextAreaRow { row in
        row.placeholder = "本文を入力"
    }

    // 入力用のセクションを作ってRowを追加
    let section = Section()
    section.append(contentsOf: [
        titleRow,
        bodyRow
    ])

    form.append(contentsOf: [
        section
    ])
}
```

ナビゲーションバーの「Save」をクリックすると内容をアラートで表示します。

サンプルプログラム　Bonus/Chapter4/EurekaExample/EurekaExample/ViewController.swift

```
@IBAction func onTap(_ sender: Any) {
    // タイトルを取得
    let title = (form.allRows[0] as? TextRow)?.value ?? ""
    // 本文を取得
    let body = (form.allRows[1] as? TextAreaRow)?.value ?? ""
    let message = """
                    title: \(title)
                    body: \(body)
                  """
    let alertController = UIAlertController(
```

```
        title: nil,
        message: message,
        preferredStyle: .alert
    )
    alertController.addAction(UIAlertAction(title: "OK",
                                            style: .default,
                                            handler: nil))
    present(alertController, animated: true, completion: nil)
}
```

このような数行のプログラムを書くことでフォームを表示させることができます。

Swiftは独自の演算子を定義できます。Eurekaではそれが活用されており先ほどと同じフォームを以下のようなプログラムでも書くことができます。

サンプルプログラム　Bonus/Chapter4/EurekaCustomOperator/EurekaCustomOperator/ViewController.swift

```
let titleRow = TextRow { row in
    row.title = "タイトル"
    row.placeholder = "タイトルを入力"
}

let bodyRow = TextAreaRow { row in
    row.placeholder = "本文を入力"
}

let section = Section()
section
    <<< titleRow
    <<< bodyRow

form +++ section
```

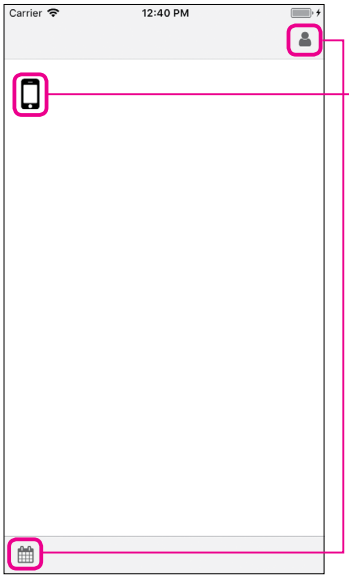
どちらが読みやすいかは好みによるでしょう。

筆者は独自演算子を学習するコストを考慮して、基本的には使わない方針にしています。

Section 4.5

アイコンフォント —SwiftIconFont

サンプルアプリ



画像を使わずにアイコン付きの各種ボタンを表示します。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/SwiftIconFontExample.xcodeproj
Storyboard	Bonus/Chapter4/SwiftIconFontExample/SwiftIconFontExample/Base.lproj/Main.storyboard
ソース	Bonus/Chapter4/SwiftIconFontExample/SwiftIconFontExample/ViewController.swift

SwiftIconFontについて

アプリのボタンなどにアイコンを使いたいことがあります。

標準でいくつかのアイコン素材が提供されていますが、充分ではないこともあります。

画像のアイコン素材を使うと、色やサイズの変更などに手間がかかります。そんな時に便利なのがアイコンフォントです。文字としてアイコンを表示できるので、色やサイズなどを柔軟に指定することができます。

ここでは **SwiftIconFont**（https://github.com/0x73/SwiftIconFont）でアイコンを表示させてみます。

サンプルアプリの作成

1. Cartfileの作成

Cartfileに次の内容を記載します。

サンプルプログラム　Bonus/Chapter4/SwiftIconFontExample/Cartfile

```
github "0x73/SwiftIconFont" == 2.7.2
```

2. ビルド

次のようにビルドします。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

```
*** Cloning SwiftIconFont
*** Checking out SwiftIconFont at "2.7.2"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.hM2LCS.log
*** Building scheme "SwiftIconFont" in SwiftIconFont.xcodeproj
```

3. Linked Frameworks and Librariesの追加

Carthage/Build/iOS/SwiftIconFont.frameworkをLinked Frameworks and Librariesにドラッグ&ドロップします。

4. Run Script

次のように設定します。

スクリプト

```
/usr/local/bin/carthage copy-frameworks
```

Input Files

```
$(SRCROOT)/Carthage/Build/iOS/SwiftIconFont.framework
```

Output Files

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/SwiftIconFont.framework
```

5. アイコンの表示

SwiftIconFontをimportします。

サンプルプログラム　Bonus/Chapter4/SwiftIconFontExample/SwiftIconFontExample/ViewController.swift

```
import SwiftIconFont
```

viewDidLoadでアイコンを設定したLabelをViewに追加します。ナビゲーションバーやツールバーにも表示します。

サンプルプログラム　Bonus/Chapter4/SwiftIconFontExample/SwiftIconFontExample/ViewController.swift

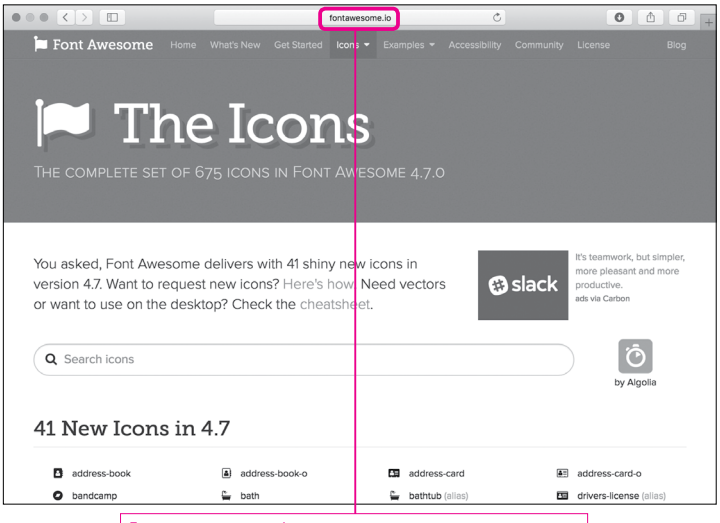
```
override func viewDidLoad() {
    super.viewDidLoad()

    // スマートフォンのアイコンを持つラベル
    let label = UILabel(
        frame: CGRect(x: 20.0, y: 80.0, width: 50.0, height: 50.0)
    )
    label.font = UIFont.icon(from: .FontAwesome, ofSize: 50.0)
```

```
    label.text = String.fontAwesomeIcon("mobile")
    view.addSubview(label)

    // ユーザーのアイコンを持つナビゲーションバーアイテム
    let navItem = UIBarButtonItem()
    navItem.icon(from: .FontAwesome, code: "user", ofSize: 20.0)
    navigationItem.rightBarButtonItem = navItem

    // カレンダーのアイコンを持つツールバーアイテム
    let toolItem = UIBarButtonItem()
    toolItem.icon(from: .FontAwesome, code: "calendar", ofSize: 20.0)
    toolbarItems = [toolItem]
}
```

このように画像を使わずアイコンを簡単に表示することができます。

Note

ここでは FontAwesomeというフォントを使いました。次のページで FontAwesomeフォントで利用できるアイコンの一覧を確認することができます。

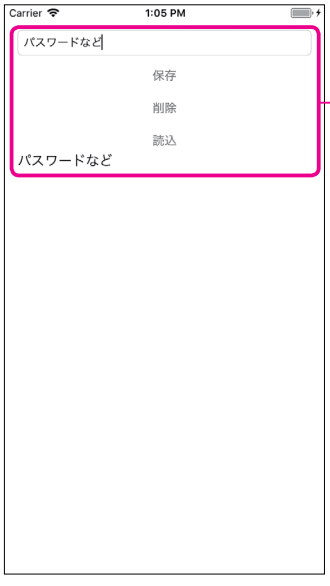


「FontAwewome」http://fontawewome.io/icons/

Section 4.6

機密情報保存—KeychainAccess

サンプルアプリ



入力したテキストを保存・読込・削除します。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/KeychainAccessExample.xcodeproj
Storyboard	Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/Base.lproj/Main.storyboard
ソース	Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

KeyChainAccessについて

書籍版のChapter 18で紹介したUserDefaultsやファイルにそのまま保存する方法では情報が暗号化されません。そのため機密情報、例えばパスワードなどの認証情報をその方法で保存するのは安全とはいえません。

iOSにはキーチェーンというセキュアな情報保存の仕組みが用意されているのですが、とても複雑でこれを自分で操作するのは大変です。

KeychainAccess（https://github.com/kishikawakatsumi/KeychainAccess）というライブラリを使うと簡単にキーチェーンに情報を保存できます。

サンプルアプリの作成

1. Cartfileの作成

次の内容をCartfileに記載します。

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/Cartfile

```
github "kishikawakatsumi/KeychainAccess" == 3.1.0
```

2. ビルド

次のようにビルドします。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

```
*** Cloning KeychainAccess
*** Checking out KeychainAccess at "v3.1.0"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.cF1gkv.log
*** Building scheme "KeychainAccess" in KeychainAccess.xcworkspace
```

3. Linked Frameworks and Librariesの追加

Carthage/Build/iOS/KeychainAccess.frameworkを Linked Frameworks and Libraries にドラッグ&ドロップします。

4. Run Script

次のように設定します。

スクリプト

```
/usr/local/bin/carthage copy-frameworks
```

Input Files

```
$(SRCROOT)/Carthage/Build/iOS/KeychainAccess.framework
```

Output Files

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/KeychainAccess.framework
```

5. 機密データの保存

キーチェーンに情報を保存してすぐ読み出すだけの単純な処理を行います。

KeychainAccessを`import`します。

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

```
import KeychainAccess
```

`viewDidLoad`でパスワードを保存します。

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

```
// 複数のメソッドで使うのでプロパティとしてインスタンスを作成
let keychain = Keychain()

@IBAction func onTapSave(_ sender: Any) {
    guard let text = textField.text, !text.isEmpty else { return }
    keychain["secret"] = text // "secret"をキーとしてテキストを保存
}
```

viewDidAppearで先程保存したパスワードを読み込んでラベルに表示します。

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

```
@IBAction func onTapLoad(_ sender: Any) {
    // "secret"をキーとして保存された情報を取得
    label.text = keychain["secret"]
}
```

nilを代入すれば削除できます。

また、以下のようにremoveメソッドを使っても、保存した情報を削除することができます。

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

```
keychain.remove("secret")
```

サンプルプログラム　Bonus/Chapter4/KeychainAccessExample/KeychainAccessExample/ViewController.swift

```
@IBAction func onTapRemove(_ sender: Any) {
    keychain["secret"] = nil // "secret"をキーと保存された情報を削除
}
```

UserDefaultsと変わらない手軽さですが、これで情報を安全に保存することができます。

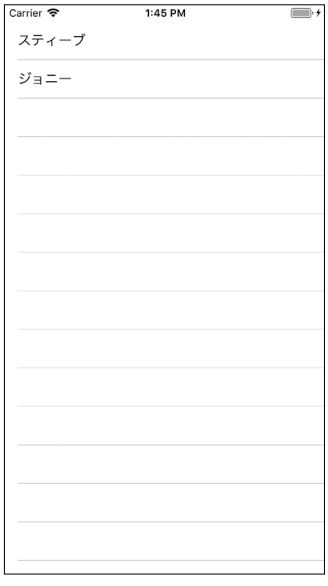
Note

KeychainAccessライブラリは、この他にもiCloudによる同期や、Touch IDによる認証などもサポートしています。

Section 4.7

データベース—Realm

サンプルアプリ



サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/RealmExample.xcodeproj
Storyboard	Bonus/Chapter4/RealmExample/RealmExample/Base.lproj/Main.storyboard
ソース	Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

Realmについて

アプリが扱うデータの量が多くなってくると、UserDefaultsやファイルへの保存だけでは管理が大変になってきます。また検索などの機能も用意されていないため、これらの方法はそもそも大量のデータを管理するには向いていません。そこでデータベースを使うことになります。

iOSには Core Dataという標準のデータベースがありますが、利用する手順が複雑です。ここでは簡単に使える **Realm**（https://github.com/realm/realm-cocoa）を紹介します。Realmには Androidなど iOS以外のプラットフォームでも動くという特徴があります。

サンプルアプリの作成

1. Cartfileの作成

次の内容をCartfileに記載します。

サンプルプログラム　Bonus/Chapter4/RealmExample/Cartfile

```
github "realm/realm-cocoa" == 3.0.1
```

2. ビルド

次のようにビルドします。ここまではこれまでと同様の手順です。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ carthage update --platform iOS↵
```

実行結果

```
*** Fetching realm-cocoa
*** Downloading realm-cocoa.framework binary at "v3.0.1"
*** xcodebuild output can be found in /var/folders/xx/2bhj86b17j9ch
hv6wr_h7d4h0000gp/T/carthage-xcodebuild.6xR4B2.log
```

3. Linked Frameworks and Librariesの追加

Carthage/Build/iOS/RealmSwift.frameworkとCarthage/Build/iOS/Realm.frameworkをLinked Frameworks and Librariesにドラッグ&ドロップします。

2つのフレームワークが必要なので注意してください。

4. Run Script

以下のように設定します。ここも2つのフレームワークを指定する必要があります。

スクリプト

```
/usr/local/bin/carthage copy-frameworks
```

Input Files

```
$(SRCROOT)/Carthage/Build/iOS/Realm.framework
$(SRCROOT)/Carthage/Build/iOS/RealmSwift.framework
```

Output Files

```
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/Realm.framework
$(BUILT_PRODUCTS_DIR)/$(FRAMEWORKS_FOLDER_PATH)/RealmSwift.framework
```

5. データベースの利用

ユーザー情報を2人分保存してそれを一覧表示するサンプルです。

importします。インポートは1つで大丈夫です。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

```
import RealmSwift
```

テーブルビューコントローラーを継承します。Main.storyboard でも元々配置されている View Controllerの代わりに Table View Controllerを使います。詳しい手順は書籍版の Chapter 13を参照してください。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

```
class ViewController: UITableViewController {
```

データーベースから取得した結果を保存する変数を用意します。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

```
// データベースから取得したPerson一覧を保持するプロパティ
var people: Results<Person>!
```

viewDidLoadではデータベースからユーザーを全件取得し、結果が空なら 2件表示する処理を行っています。この処理のあとTableViewを更新するためにreloadDataを呼んでいます。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()

    let realm = try! Realm()              // Realmのインスタンスを作成
    people = realm.objects(Person.self) // Person一覧を取得

    if people.isEmpty {                   // Person一覧が空なら
        let person1 = Person()
        person1.name = "スティーブ"
        let person2 = Person()
        person2.name = "ジョニー"

        try! realm.write {                // 2人のPersonを追加
            realm.add(person1)
            realm.add(person2)
        }
    }

    tableView.reloadData()                // TableViewを再読込
}
```

TableViewに表示するためのDataSource、Delegateの設定も必要です。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/ViewController.swift

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return people.count // 一覧の数だけセルを表示
}
```

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "reuseIdentifier", for: indexPath
    )

    // 名前をセルに表示
    let person = people[indexPath.row]
    cell.textLabel?.text = person.name

    return cell
}
```

保存するデータはこのように定義します。

サンプルプログラム　Bonus/Chapter4/RealmExample/RealmExample/Person.swift

```
import Foundation
import RealmSwift

class Person: Object {
    @objc dynamic var name = ""
}
```

単純な例ですが、データベースへの保存、データベースからの読み込みと表示ができました。

Realmに保存するデータは、`Object`クラスを継承する必要があります。

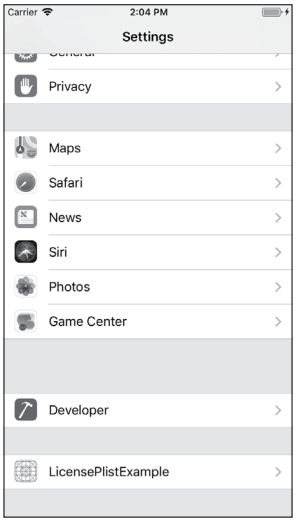
また、プロパティには「`@objc dynamic`」を付けるのを忘れないようにしましょう。これによってRealmがプロパティを管理できるようになります。

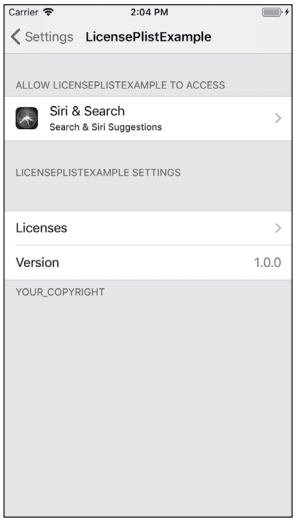
Note

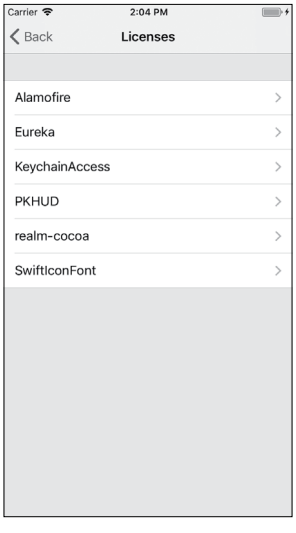
Realmは、検索などデータベースに必要な様々な機能を提供しています。
少し情報は遅くなりますが、日本語ドキュメントも用意されていますので、活用を検討してみてください。
https://realm.io/jp/docs/swift/latest/

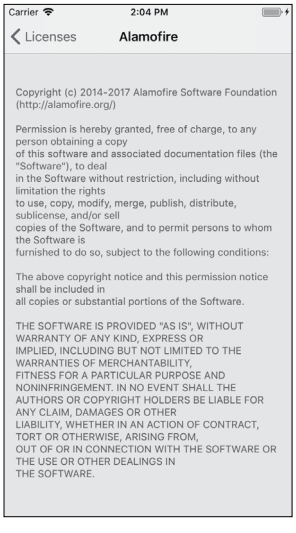
Section 4.8 ライセンスの表示—LicensePlist

サンプルアプリ









iOSの「設定」アプリにライセンス情報を表示させます。

サンプルファイル

種類	ファイル
プロジェクト	Bonus/Chapter4/RealmExample.xcodeproj
Storyboard	変更しません
ソース	変更しません

LicensePlistについて

ここで紹介したライブラリは、すべてオープンソースライセンスに基づき公開されています。とはいえ、無条件で使えるわけではありません。各ライブラリのライセンスに従う必要があります。

ここで使用したライブラリとそれらのライセンスは次のとおりです。

ライブラリ	ライセンス
Alamofire	MIT
PKHUD	MIT
Eureka	MIT
SwiftIconFont	MIT
KeychainAccess	MIT
Realm	Apache 2.0

いずれも比較的、制約の少ないライセンスです。使用しているライブラリとそのライセンスを閲覧できるようにしておけば基本的に問題ないでしょう。

`LicensePlist`（https://github.com/mono0926/LicensePlist）を使えば、iOSの「設定」アプリにライセンス一覧を簡単に表示させることができます。

Note

ライブラリをアプリに使用する際には必ずライセンスを確認しましょう。
有志の方々の手によって、以下に各ライセンスの日本語訳が公開されています。

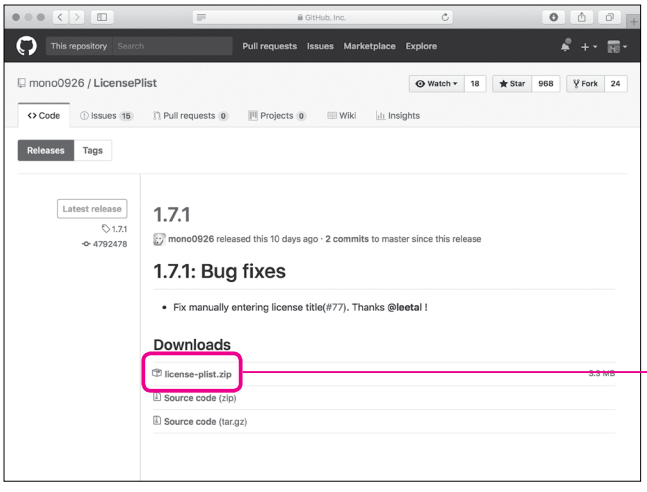
●The MIT License
https://ja.osdn.net/projects/opensource/wiki/licenses%2FMIT_license

●Apache License, Version 2.0
https://ja.osdn.net/projects/opensource/wiki/licenses%2FApache_License_2.0

インストール

LicensePlistは次のサイトからダウンロードすることができます。

https://github.com/mono0926/LicensePlist/releases



zipが展開されていない場合はダブルクリックして license-plistファイルをダウンロードフォルダに格納しておきます。(Safariがデフォルト設定の場合は自動で展開されます。)

続いて、次のコマンドを実行します。コマンドを実行するディレクトリーはどこでも構いません。

ターミナル

```
$ sudo cp ~/Downloads/license-plist /usr/local/bin/↵
Password: Macの管理者パスワード↵
```

実行結果

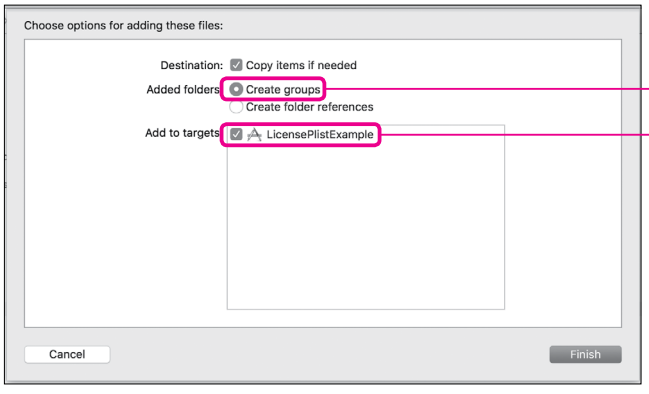
```
なし（エラーメッセージが表示されないこと）
```

Settings.bundleの準備

iOSの「設定」アプリへ情報を表示させるには`Settings.bundle`というファイルが必要です。

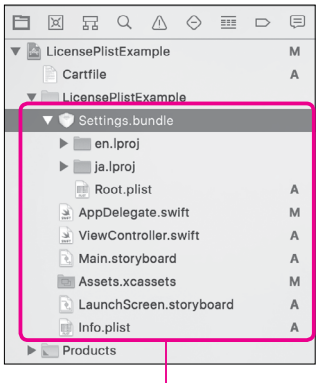
次のサイトからダウンロードして展開した `Settings.bundle`をドラッグ&ドロップしてプロジェクトに取り込みます。

https://github.com/mono0926/LicensePlist/raw/master/Settings.bundle.zip

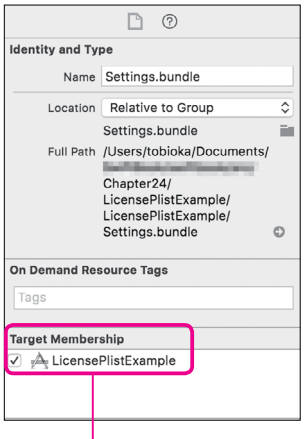


「Create groups」を選択します。

全てチェックONにします。



このようなファイル構成になります。



「File inspector」で「Target Membership」の「LicensePlistExample」にチェックが入っていることを確認します。

ライセンス一覧の追加

次のコマンドでライセンス一覧を作成できます。

ターミナル

```
$ cd プロジェクトフォルダーの場所↵
$ license-plist↵
```

この Sectionのサンプルプロジェクトに含まれる Carfileには、これまでに紹介した全てのライブラリを含めてあります。LicensePlistはこれらを基にライセンス情報を生成してくれます。

サンプルプログラム　Bonus/Chapter4/LicensePlistExample/Cartfile

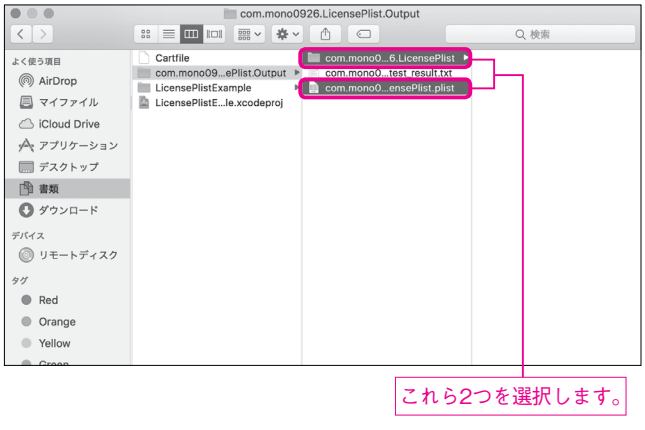
```
github "Alamofire/Alamofire" == 4.5.1
github "pkluz/PKHUD" == 5.0.0
github "xmartlabs/Eureka" == 4.0.1
github "0x73/SwiftIconFont" == 2.7.2
github "kishikawakatsumi/KeychainAccess" == 3.1.0
github "realm/realm-cocoa" == 3.0.0
```

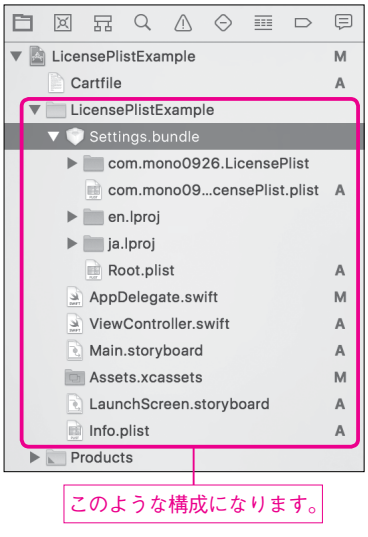
実行結果

```
[2017-10-15T13:58:10.450+09:00] [WARNING] Not found: license_plist.yml
-- file:///Users/tobioka/Documents/Bonus/Chapter4/LicensePlistExample/.
[2017-10-15T13:58:10.455+09:00] [INFO] Start
[2017-10-15T13:58:10.456+09:00] [WARNING] not found: Pods/Target%20Supp
ort%20Files -- file:///Users/tobioka/Documents/Bonus/Chapter4/LicensePl
istExample/
[2017-10-15T13:58:10.456+09:00] [INFO] Pods License parse start
[2017-10-15T13:58:10.456+09:00] [WARNING] Not found: Pods/Manifest.lock
 -- file:///Users/tobioka/Documents/Bonus/Chapter4/LicensePlistExample/.
[2017-10-15T13:58:10.459+09:00] [WARNING] Not found: Cartfile.resolved
-- file:///Users/tobioka/Documents/Bonus/Chapter4/LicensePlistExample/.
[2017-10-15T13:58:10.461+09:00] [INFO] Carthage License collect start
[2017-10-15T13:58:10.462+09:00] [INFO] Manual License start
[2017-10-15T13:58:10.462+09:00] [WARNING] Not found: com.mono0926.Licen
sePlist.Output/com.mono0926.LicensePlist.latest_result.txt -- file:///U
sers/tobioka/Documents/Bonus/Chapter4/LicensePlistExample/.
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: Al
amofire, name: Alamofire)
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: pk
luz, name: PKHUD)
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: xm
artlabs, name: Eureka)
```

```
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: 0x
73, name: SwiftIconFont)
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: ki
shikawakatsumi, name: KeychainAccess)
[2017-10-15T13:58:10.463+09:00] [INFO] license download start(owner: re
alm, name: realm-cocoa)
[2017-10-15T13:58:12.560+09:00] [INFO] Directory created: com.mono0926.
LicensePlist.Output -- file:///Users/tobioka/Documents/Bonus/Chapter4/L
icensePlistExample/
[2017-10-15T13:58:12.566+09:00] [INFO] End
[2017-10-15T13:58:12.566+09:00] [INFO] ----------Result----------
[2017-10-15T13:58:12.566+09:00] [INFO] # Missing license:
[2017-10-15T13:58:12.566+09:00] [INFO] None
```

作成されたcom.mono0926.LicensePlist.Output内のcom.mono0926.LicensePlist.plistとcom.mono0926.LicensePlistを先程のSettings.bundle内にドラッグ＆ドロップして追加します。





この状態でアプリを実行します。シミュレータの「設定」アプリを開くと、ライセンス一覧が表示されます。

Note

en.lproj（英語）、ja.lproj（日本語）のRoot.stringsを編集するとコピーライトの表記などを変更可能です。

Section 4.9 ソースコードのスタイルチェック

これまでのライブラリはアプリの機能を直接改善するものでしたが、最後に番外編として開発を支援するツールを紹介します。

ある処理を行わせるためのプログラムの書き方は１つとは限りません。そのため、時間をかけて開発したり、複数の人が協力して開発したりする場合には、プログラムの書き方を統一するルールを決めるのが一般的です。

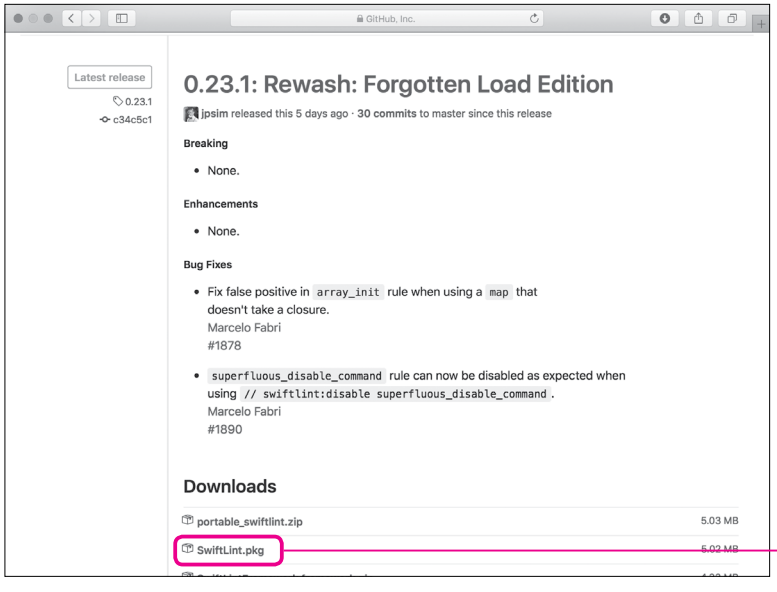
こうしたルールを意識せずにプログラムを書き進めると、時間の経過とともに一貫性が失われてゆき、次第に読みづらくなります。

SwiftLint（https://github.com/realm/SwiftLint）というツールを使うと、プログラムの記述スタイルを自動的にチェックしてもらえるようになります。読みづらい箇所や、一貫性が失われている箇所などを自動的に探し出して、指摘してくれるため、結果的に開発速度も向上します。

インストール

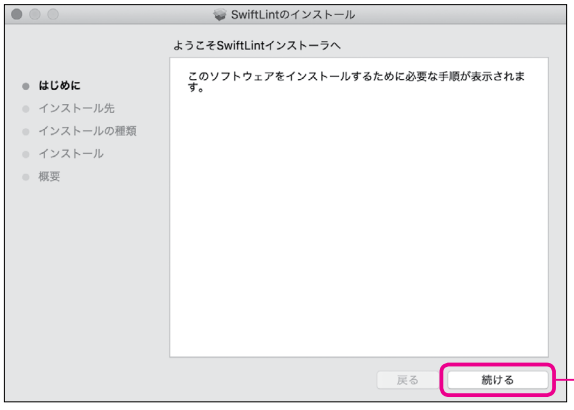
SwiftLintは次のサイトからダウンロードすることができます。

https://github.com/realm/SwiftLint/releases



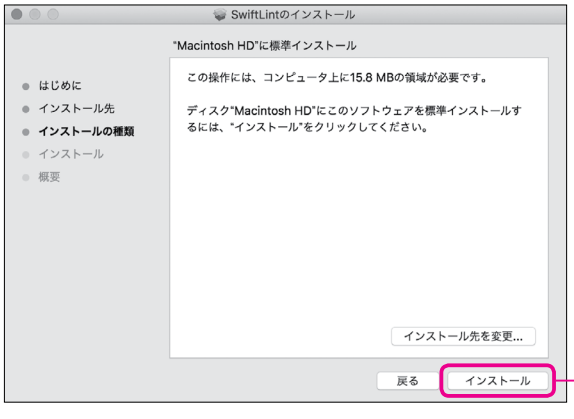
「SwiftLint.pkg」をクリックします。

インストーラーの指示に従ってインストールします。

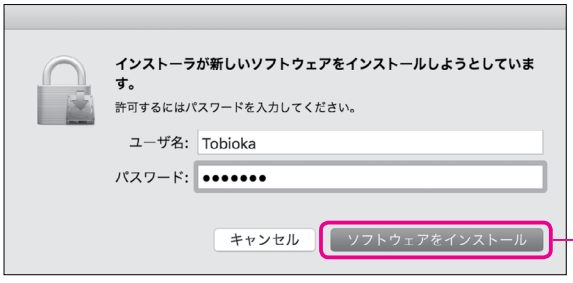


「続ける」をクリックします。

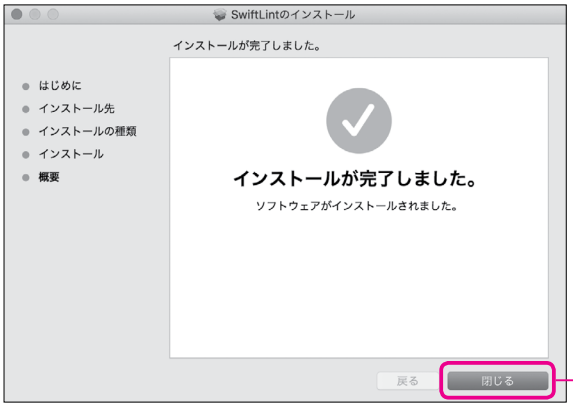
「続ける」をクリックします。



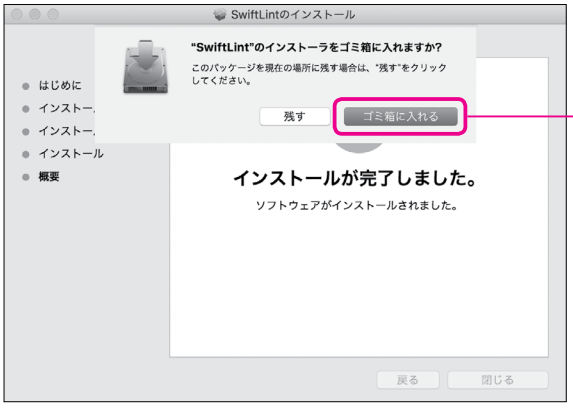
「インストール」をクリックします。



「ソフトウェアをインストール」をクリックします。



「閉じる」をクリックします。



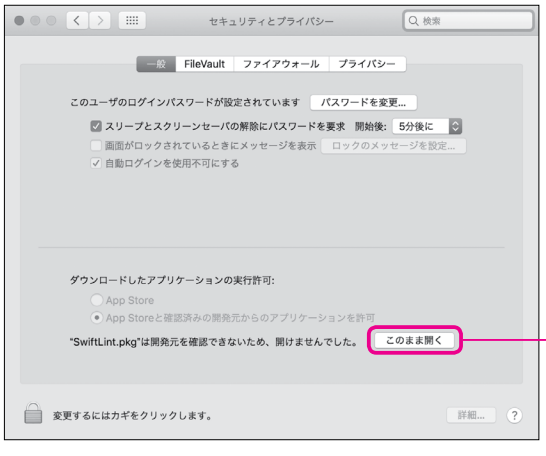
「ゴミ箱に入れる」をクリックします。

セキュリティ設定

Carthageの時と同様、開発元が未確認となってしまうため、インストールを許可する必要があります。

「OK」をクリックして閉じます。

「セキュリティとプライバシー」をクリックします。



「このまま開く」をクリックします。

Note

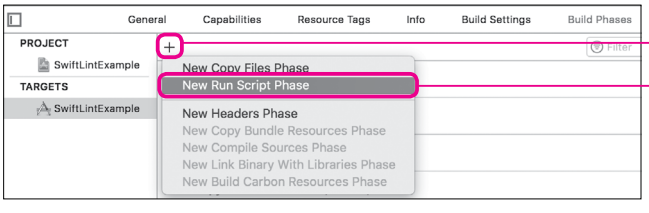
Carthageと同様、セキュリティを弱める操作になるので注意して行なってください。

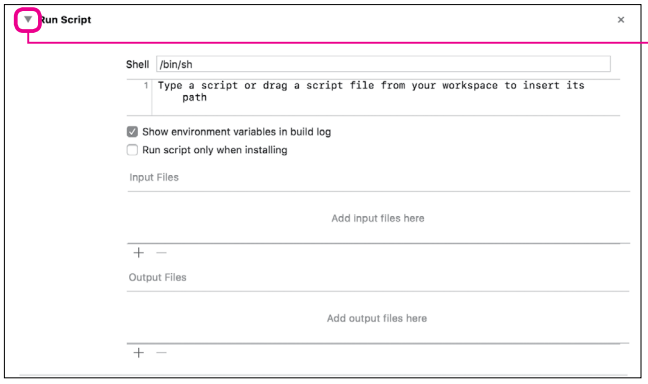
Run Script

次のように設定します。

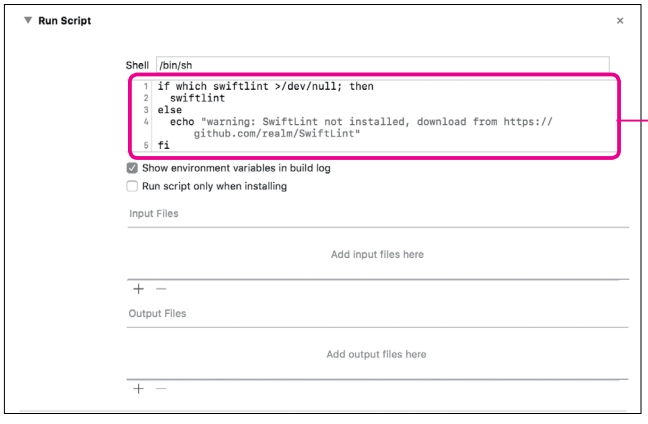
スクリプト

```
if which swiftlint >/dev/null; then
  swiftlint
else
  echo "warning: SwiftLint not installed, download from https://github.
com/realm/SwiftLint"
fi
```





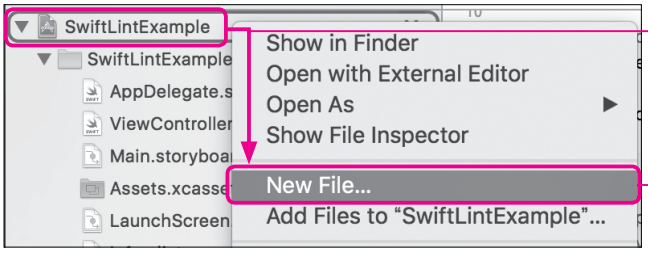
「Run Script」を開きます。



スクリプトの内容を入力しました。

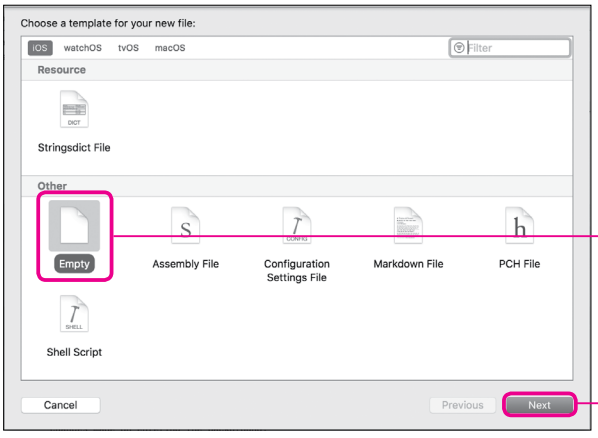
設定

プロジェクトフォルダーに`.swiftlint.yml`というファイルを作成します。ここにチェックの設定を書きます。



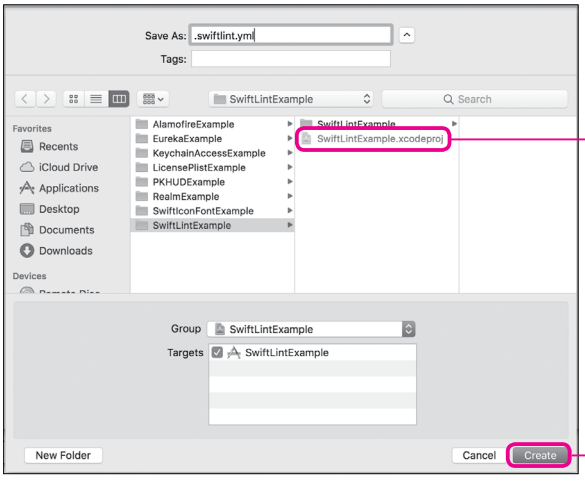
①右クリックします。

②「New File...」を選択します。



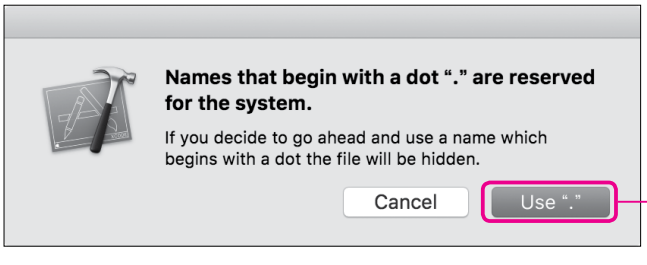
①「Empty」を選択します。

②「Next」をクリックします。



①「.xcodeproj」ファイルがあるフォルダーを選択します。

②「Create」をクリックします。



「Use "."」をクリックします。

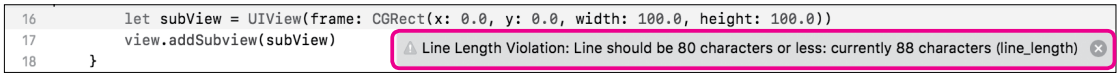
サンプルプログラム　Bonus/Chapter4/SwiftLintExample/.swiftlint.yml

```
excluded:
- Carthage
line_length: 80
```

ここではCarthageで入れたライブラリのコードを除外し、1行の文字数を80文字にしています。

チェックの実施

アプリをビルド、実行した場合に、プログラムが自動的にチェックされます。



行の長さが長すぎると指摘されています。

ルールの無効化

SwiftLintの指摘を無効化したい場合は、`.swiftlint.yml`に以下のような設定をします。

サンプルプログラム　swiftlint.yml

```
disabled_rules:
  - line_length
```

この設定によって行が長くても警告が表示されなくなります。たとえば既存のプロジェクトに途中から導入して、警告が多すぎるためひとまず無視したい、という時に有効でしょう。

また、よくない書き方だとわかっているが、特定の箇所ではやむを得ず指摘を無視したいということもあるでしょう。

そのような場合は以下のようなコメントを書くことで無効化できます。

サンプルプログラム　swiftlint.yml

```
// swiftlint:disable:next line_length
```

このコメントが書かれた次の行は、長さによる警告は行われなくなります。

ここでは便利なライブラリの使い方を紹介しました。これらをぜひ活用して、高機能なアプリを効率的に開発してみてください。

以上でボーナスPDFは終わりです。詳細は書籍『Swift 4プログラミング入門 iOS 11＋Xcode 9対応』をご購入ください。

Amazonで購入

書　　名	Swift 4プログラミング入門 iOS 11＋Xcode 9対応
著　　者	飛岡辰哉
発売開始日	2018年３月19日
発 行 者	ソシム株式会社 http://www.socym.co.jp/
定　　価	3,564円（本体3,300円＋税）
ＩＳＢＮ	978-4-8026-1153-4
判　　型	B5変型判
ページ数	800ページ

サンプルプログラムのダウンロード

著者 飛岡 辰哉 とびおか たつや

HOME　Twitter　Facebook　GitHub

渋谷区在住のソフトウェアエンジニア。iOSアプリの開発は2010年に始め、代表作のJSAnywhereは、Apple社によるApp Store上での特集や雑誌にも掲載される。その他にも様々なアプリをリリースし続けている。