clearbridge mobile

How To Do More With Less:

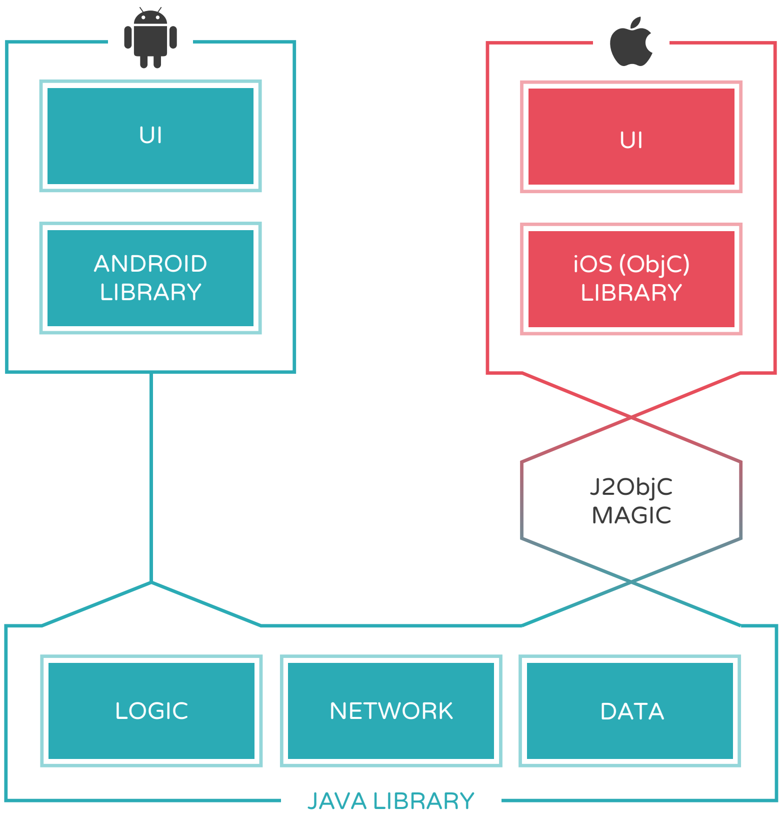
J2ObjC & Cross-Platform App Development

AN INTRODUCTION TO J2OBJC

This whitepaper will provide an overview of J2ObjC and its application in cross-platform development. It will cover what J2ObjC is, how it functions, its advantages and disadvantages, and use cases of J2ObjC projects Clearbridge Mobile has completed. It will also convey why J2ObjC is an effective tool for developing cross-platform applications, both from a development and a business perspective.

WHAT IS J2OBJC

J2ObjC is an open-source command-line tool that was created by Google to solve code duplication in cross-platform development (j2objc.org). It allows developers to write all the business logic and data model handling in Java and then convert it to Objective-C for the iOS platform, without having to code twice (UI code must be written natively outside of this core, as J2ObjC does not provide any platform-independent UI toolkit). Using J2ObjC allows for the reduction of duplicate code in cross-platform development without sacrificing native functionality on either Android or iOS.



BENEFITS OF J2OBJC

J2ObjC has many benefits, including:

- Reduced duplicate code
- Shorter development cycles
- Enforcement of code architecture (Model, View, Controller)
- Separation of UI and business logic
- Reduced business logic inconsistencies
- Fewer errors
- Shared base class definitions
- Retention of native functionality
- Improved workflow
- Easier backward and forward OS compatibility
- Faster changes, debugging, and maintenance

REDUCED DUPLICATE CODE

Since J2ObjC allows you to write business logic only once, it can reduce duplicate code by up to 70-80%. Only UI code must be written separately in Java and Objective-C outside the core.

SHORTER DEVELOPMENT CYCLES

Put simply, J2ObjC allows you to do more, faster, with fewer people in comparison to developing separately for each platform. Since the development is broken down into smaller chunks, it becomes easier to understand and therefore quicker to implement.

ENFORCEMENT OF CODE ARCHITECTURE

J2ObjC forces development teams to strictly follow MVC architecture, which separates the business logic and the UI into the "core" and the "platform." The core handles network calls, processing network responses, field validation, and app navigation. The platform is only responsible for the UI elements, providing View Controllers and implementing the interfaces that the core provides.

FEWER ERRORS

When code is decoupled it forces teams to think about the architecture and plan thoroughly, which results in adherence to best practices. There are fewer instances of sloppy code or corner-cutting, leading to fewer bugs. Furthermore, the code separation makes it easier to track down the origin of bugs that do make their way into the code so they can be fixed. If the bug related to logic, validation, or data handling, then the core is responsible. If the bug is visual, then the platform is responsible.

NATIVE FEEL & NO PERFORMANCE LOSS

With other cross-platform development tools, including HTML5 and PhoneGap, there is some loss of native functionality. This is due to extra layers of communication required to map methods to the native platform, which can affect performance. When building apps with J2ObjC, developers create the shared logic in Java on the Android side, meaning Android can directly access the methods. The logic is then transpiled to Objective-C using J2ObjC, so the iOS side has the methods directly accessible as well. Furthermore, since J2ObjC still requires the UI to be coded natively, the user experience doesn't suffer on either platform.

IMPROVED WORKFLOW

The separation of code allows developers working on the UI to continue to do so even when the core is unfinished, and vice versa. This also allows for the reallocation of resources (UI developers and core developers can help each other out) if necessary. The improved workflow helps to reduce development cycles and cut down coding time.

DISADVANTAGES OF J2OBJC

J2ObjC is far from perfect. Considering that it's a newer tool, some downsides can be expected. Current disadvantages to J2ObjC include:

- » Core can be a bottleneck
- » More effort to update core on iOS
- » Long compile times on iOS
- » Enums, methods, and classes can have long names when converted to Objective-C if proper structure is not followed

CORE BOTTLENECK

If not enough resources have been allocated to the core, it can eventually become a bottleneck for both platforms. The bottleneck happens when the UI is fully built out, but the core doesn't have all the business logic fully completed. This prevents proper end to end testing of the functionality and the UI with real data. In certain cases, this can also affect the navigation of the app because the logic for navigation doesn't yet exist.

This issue can be resolved by reallocating resources. If a platform developer is blocked by the missing functionality in the core, they can jump in and add the functionality.

UPDATES AND COMPILE TIMES ON IOS

On iOS, it takes longer to update the core because it takes time to transpile the code. When you make changes, you must clean the project and then recompile it. During the compilation on iOS, it is necessary to run a build script that will go through every Java file and transpile it to Objective-C, which takes time because of the complexity of the operation. It also requires a little more setup to get it up and running. You must have a reference to the Java source files inside your iOS project; you must add the build script; and you must also add all of the correct search paths for J2ObjC.

That said, it's possible to improve this workflow with an automation server that runs a transpile daemon. With the automation server, the need for including the Java source files and adding the build script is removed, because the daemon handles the transpile process. The daemon monitors the changes to the core Java codebase by periodically fetching from GIT and comparing the commit hashes. If a change is detected, the Java code is transpiled to Objective-C and the results are pushed to a separate branch in iOS project GIT. The iOS developer can then merge the transpiled core into their branch.

NAMING ENUMS, METHODS, AND CLASSES

Since J2ObjC works by transpiling the Java code to Objective-C and there are major differences between the two languages, the naming of enums, classes, and methods can get messy when transpiled to Objective-C.

This can be avoided by adding annotations to methods and separating the enums and classes in different files. Use **@ObjectiveCName** annotation for complex functions to avoid long name generation on iOS.

EXAMPLE OF CLASS CONVERSION

JAVA

```
package com.clearbridgemobile.core;

import com.clearbridgemobile.core.enums.EnvironmentType;
import com.clearbridgemobile.core.managers.InterfaceManager;
import com.clearbridgemobile.core.managers.NetworkManager;
import com.google.gson.Gson;
import com.google.j2objc.annotations.ObjectiveCName;

public class CoreLib {
	private InterfaceManager interfaceManager;
	private NetworkManager networkManager;
	private Gson gson;

	private static CoreLib instance;

	private CoreLib() {
		networkManager = new NetworkManager();
		interfaceManager = new InterfaceManager();
		gson = new Gson();
	}

	public static CoreLib getInstance() {
		if (instance == null) {
			instance = new CoreLib();
		}
		return instance;
	}

	@ObjectiveCName(value = “initLib:(EnvironmentTypeEnum *)env”)
	public void initLib(EnvironmentType env) {
		networkManager.setEnvironmentType(env);
	}

	public NetworkManager getNetworkManager() {
		return networkManager;
	}

	public InterfaceManager getInterfaceManager() {
		return interfaceManager;
	}

	public Gson getGson() {
		return gson;
	}
}
```

OBJECTIVE-C HEADER

```
#ifndef _CoreLib_H_
#define _CoreLib_H_

#include “J2ObjC_header.h”

@class ComGoogleGsonGson;
@class EnvironmentTypeEnum;
@class InterfaceManager;
@class NetworkManager;

@interface CoreLib : NSObject

#pragma mark Public

- (ComGoogleGsonGson *)getGson;

+ (CoreLib *)getInstance;

- (InterfaceManager *)getInterfaceManager;

- (NetworkManager *)getNetworkManager;

- (void)initLib:(EnvironmentTypeEnum *)env OBJC_METHOD_FAMILY_NONE;

@end

J2OBJC_EMPTY_STATIC_INIT(CoreLib)

FOUNDATION_EXPORT CoreLib *CoreLib_getInstance();

J2OBJC_TYPE_LITERAL_HEADER(CoreLib)

typedef CoreLib ComClearbridgemobileCoreCoreLib;

#endif // _CoreLib_H_
```

OBJECTIVE-C IMPLEMENTATION

```
#include “CoreLib.h”
#include “EnvironmentType.h”
#include “Gson.h”
#include “IOSClass.h”
#include “InterfaceManager.h”
#include “J2ObjC_source.h”
#include “NetworkManager.h”
#include “com/google/j2objc/annotations/ObjectiveCName.h”

@interface CoreLib () {
 @public
  InterfaceManager *interfaceManager_;
  NetworkManager *networkManager_;
  ComGoogleGsonGson *gson_;
}

- (instancetype)init;

@end

J2OBJC_FIELD_SETTER(CoreLib, interfaceManager_, InterfaceManager *)
J2OBJC_FIELD_SETTER(CoreLib, networkManager_, NetworkManager *)
J2OBJC_FIELD_SETTER(CoreLib, gson_, ComGoogleGsonGson *)

static CoreLib *CoreLib_instance_;
J2OBJC_STATIC_FIELD_GETTER(CoreLib, instance_, CoreLib *)
J2OBJC_STATIC_FIELD_SETTER(CoreLib, instance_, CoreLib *)

__attribute__((unused)) static void CoreLib_init(CoreLib *self);

__attribute__((unused)) static CoreLib *new_CoreLib_init() NS_RETURNS_RETAINED;

@implementation CoreLib

- (instancetype)init {
  CoreLib_init(self);
  return self;
}

+ (CoreLib *)getInstance {
  return CoreLib_getInstance();
}
```

```
- (void)initLib:(EnvironmentTypeEnum *)env {
   [((NetworkManager *) nil_chk(networkManager_))
 setEnvironmentTypeWithEnvironmentTypeEnum:env];
 }

- (NetworkManager *)getNetworkManager {
   return networkManager_;
 }

- (InterfaceManager *)getInterfaceManager {
   return interfaceManager_;
 }

- (ComGoogleGsonGson *)getGson {
   return gson_;
 }

 + (IOSObjectArray *)__annotations_initLibWithEnvironmentTypeEnum_ {
   return [IOSObjectArray arrayWithObjects:(id[]) {
 [[ComGoogleJ2objcAnnotationsObjectiveCName alloc]
 initWithValue:@“initLib:(EnvironmentTypeEnum *)env”] } count:1
 type:JavaLangAnnotationAnnotation_class_()];
 }

 + (const J2ObjcClassInfo *)__metadata {
   static const J2ObjcMethodInfo methods[] = {
     { “init”, “CoreLib”, NULL, 0x2, NULL, NULL },
     { “getInstance”, NULL, “Lcom.clearbridgemobile.core.CoreLib;”, 0x9, NULL, NULL
 },
     { “initLib:”, “initLib”, “V”, 0x1, NULL, NULL },
     { “getNetworkManager”, NULL, “Lcom.clearbridgemobile.core.managers.
 NetworkManager;”, 0x1, NULL, NULL },
     { “getInterfaceManager”, NULL, “Lcom.clearbridgemobile.core.managers.
 InterfaceManager;”, 0x1, NULL, NULL },
     { “getGson”, NULL, “Lcom.google.gson.Gson;”, 0x1, NULL, NULL },
   };
   static const J2ObjcFieldInfo fields[] = {
     { “interfaceManager_”, NULL, 0x2, “Lcom.clearbridgemobile.core.managers.
 InterfaceManager;”, NULL, NULL,  },
     { “networkManager_”, NULL, 0x2, “Lcom.clearbridgemobile.core.managers.
 NetworkManager;”, NULL, NULL,  },
     { “gson_”, NULL, 0x2, “Lcom.google.gson.Gson;”, NULL, NULL,  },
     { “instance_”, NULL, 0xa, “Lcom.clearbridgemobile.core.CoreLib;”, &CoreLib_
 instance_, NULL,  },
   };
   static const J2ObjcClassInfo _CoreLib = { 2, “CoreLib”, “com.clearbridgemobile.
 core”, NULL, 0x1, 6, methods, 4, fields, 0, NULL, 0, NULL, NULL, NULL };
   return &_CoreLib;
}

@end

void CoreLib_init(CoreLib *self) {
  (void) NSObject_init(self);
  self->networkManager_ = new_NetworkManager_init();
  self->interfaceManager_ = new_InterfaceManager_init();
  self->gson_ = new_ComGoogleGsonGson_init();
}

CoreLib *new_CoreLib_init() {
  CoreLib *self = [CoreLib alloc];
  CoreLib_init(self);
  return self;
}

CoreLib *CoreLib_getInstance() {
  CoreLib_initialize();
  if (CoreLib_instance_ == nil) {
    CoreLib_instance_ = new_CoreLib_init();
  }
  return CoreLib_instance_;
}

J2OBJC_CLASS_TYPE_LITERAL_SOURCE(CoreLib)
```

HOW TO SETUP A PROJECT

There are a number of ways to set up projects using J2ObjC. Integration will vary based on the project structure you choose.

WITH AN AUTOMATION SERVER

INITIAL STEPS

1. Create 2 Git Repositories: Android, and iOS
2. Add the Git paths to the transpiling daemon on the automation server

Note: Changes to the Android repo and the Core repo will be committed separately

ANDROID

1. Create a new project, add a new module as the Core, and commit everything to the Android repo
2. Start work on the business logic in the Core inside the Android project

Android Base Project Template:
https://github.com/ClearbridgeMobile/Android-Base-Project

IOS

1. Create a new project and push it to the iOS repo
2. Create a separate branch for the automation daemon to push to
3. Add CocoaPods (optional)
4. Add the search paths for the J2ObjC Library and headers
5. Add the linker flags for J2ObjC
6. Drag prefixes.properties into the XCode project

Please refer to Image 1.

iOS Base Project Template:
https://github.com/ClearbridgeMobile/iOS-Base-Project

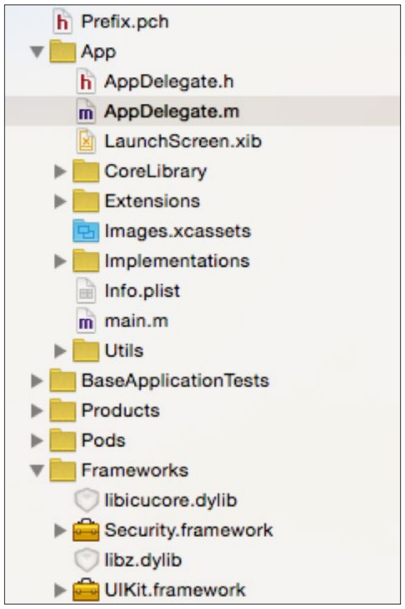


Image 1

UPDATING THE CORES

1. When including new features to core, be mindful of both platforms – updates could cause crashes if done carelessly
2. Create a separate branch for each major feature or dependency on new interfaces
3. When new feature branch is merged into dev, the Android and iOS repos should update their submodules to the correct Core commit

CORE UPDATES FOR IOS

1. Merge in the branch where the transpiled code get pushed to by the daemon
2. If new files were added then drag in the folder containing the generated Objective-C code

CORE UPDATES FOR ANDROID

1. The core is already a part of the Android project

FROM A BASE PROJECT TEMPLATE

With the automation server workflow it becomes possible to set up base project templates that can be used to start new projects. These templates contain the basic project folder structure, some useful reusable classes (network calls, local storage, etc.), and have all the flags already set for you. The only thing you have to do is change the project name and add the Git paths to the automation daemon.

WITHOUT AN AUTOMATION SERVER (GIT SUBMODULE APPROACH)

INITIAL STEPS

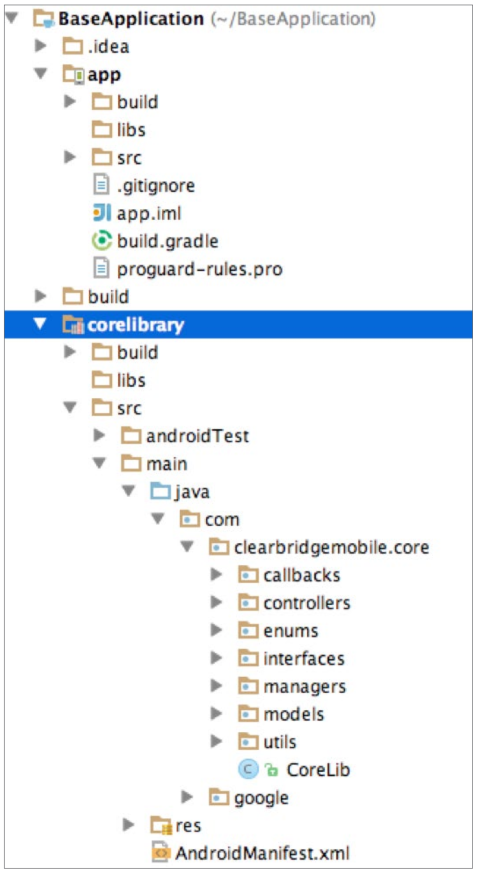
1. Create 3 Git Repositories: Android, iOS, and Core

ANDROID

1. Create a new project and commit it to the Android repo
2. In the project, add a new module as a Library from the Core
3. Move the Library into a separate folder and commit it to the Core repo
4. Add the Core submodule to Android repo (ensure the correct folder placement)
5. Start work on the business logic in the Core inside the Android project

Please refer to Image 2.

Note: Changes to the Android repo and the Core repo will be committed separately



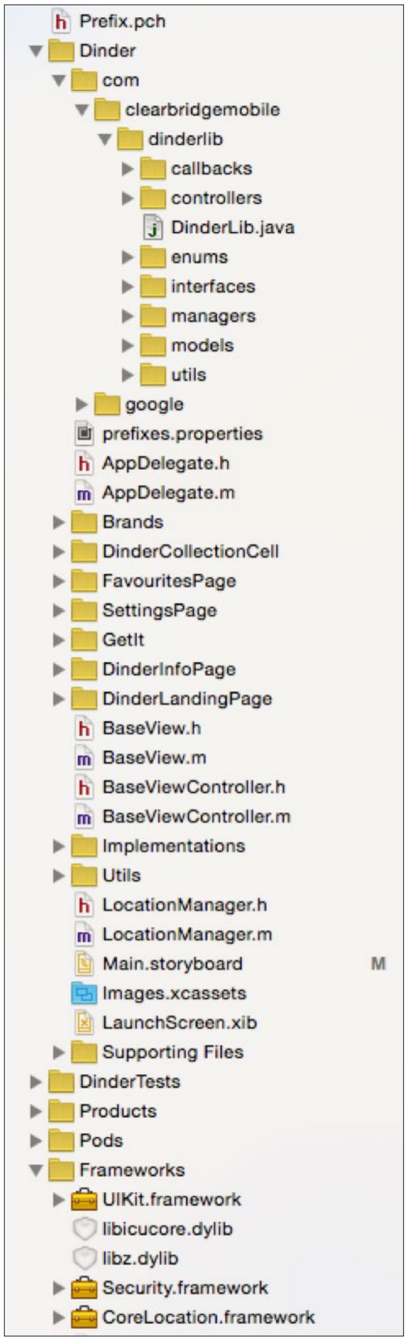
```
▼ BaseApplication (~/BaseApplication)
  ▶ .idea
  ▼ app
    ▶ build
      libs
    ▶ src
      .gitignore
      app.iml
      build.gradle
      proguard-rules.pro
  ▶ build
  ▼ corelibrary
    ▶ build
      libs
    ▼ src
      ▶ androidTest
      ▼ main
        ▼ java
          ▼ com
            ▼ clearbridgemobile.core
              ▶ callbacks
              ▶ controllers
              ▶ enums
              ▶ interfaces
              ▶ managers
              ▶ models
              ▶ utils
                CoreLib
            ▶ google
        ▶ res
          AndroidManifest.xml
```

Image 2

IOS

1. Add the Core submodule to the iOS repo (ensure the correct folder placement)
2. Add CocoaPods (optional)
3. Add the search paths for the J2ObjC Library and headers
4. Add the linker flags for J2ObjC
5. Add the custom build rule to run the Make script
6. Drag prefixes.properties into the XCode project
7. Drag the root folder for Java source code
8. Clean and build project

Please refer to Image 3.



```
  Prefix.pch
▼ Dinder
  ▼ com
    ▼ clearbridgemobile
      ▼ dinderlib
        ▶ callbacks
        ▶ controllers
          DinderLib.java
        ▶ enums
        ▶ interfaces
        ▶ managers
        ▶ models
        ▶ utils
    ▶ google
    prefixes.properties
    AppDelegate.h
    AppDelegate.m
  ▶ Brands
  ▶ DinderCollectionCell
  ▶ FavouritesPage
  ▶ SettingsPage
  ▶ GetIt
  ▶ DinderInfoPage
  ▶ DinderLandingPage
    BaseView.h
    BaseView.m
    BaseViewController.h
    BaseViewController.m
  ▶ Implementations
  ▶ Utils
    LocationManager.h
    LocationManager.m
    Main.storyboard                    M
    Images.xcassets
    LaunchScreen.xib
  ▶ Supporting Files
▶ DinderTests
▶ Products
▶ Pods
▼ Frameworks
  ▶ UIKit.framework
    libicucore.dylib
    libz.dylib
  ▶ Security.framework
  ▶ CoreLocation.framework
```

Image 3

UPDATING THE CORE

1. When including new features to core, be mindful of both platforms – updates could cause crashes if done carelessly
2. Create a separate branch for each major feature or dependency on new interfaces
3. When new feature branch is merged into dev, the Android and iOS repos should update their submodules to the correct Core commit

CORE UPDATES FOR IOS

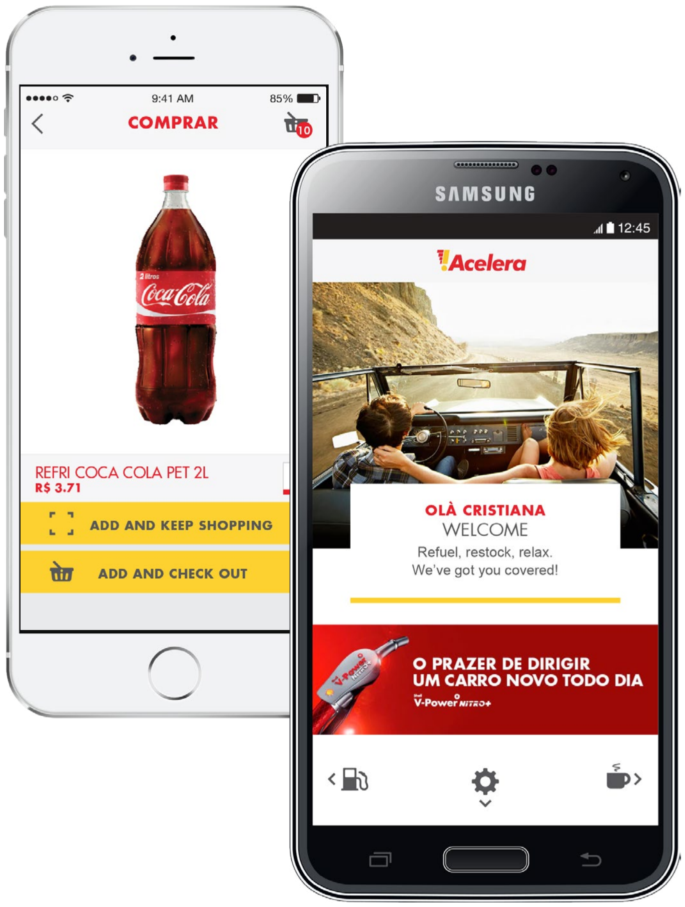
1. Pull Core
2. Remove reference to the com folder in XCode (only needed when new files are added to or deleted from the Core)
3. Drag Core com folder into XCode (only needed when new files are added to or deleted from the Core)
4. Clean and build project

CORE UPDATES FOR ANDROID

1. Pull Core
2. Clean and build project

CASE STUDY

BRINGING TOUCHLESS PAYMENTS TO GAS STATIONS



CLIENT

A multinational oil and gas company with over 4,500 stations in Brazil.

CHALLENGE

A major gas station operator in Brazil is seeing a growing threat from patrons being subject to theft when pulling out their wallets to pay for gas or in-store goods. Further, the use of stolen credit cards, known as 'dump & pump', is adding to a multi-million dollar loss in revenue.need to physically access the device.

SOLUTION

Teaming up with PayPal, Clearbridge Mobile brought to market a mobile payment solution that uses Optical Character Recognition (OCR) and GPS to accurately pay for fuel and convenience store items using a customer's smartphone – without the need to physically access the device.

The complex mobile solution, one of the first of its nature, required native development on both iOS and Android. Using J2ObjC, Clearbridge Mobile was able to build the logic for both platforms simultaneously, allowing for reduced duplicated code, fewer bugs, and a quick time to delivery.

RESULT

QUICK TIME TO DELIVERY

As a result of investing in J2ObjC, Clearbridge Mobile was able to deliver a native iOS and Android solution in under 6 weeks, versus what would have been the industry standard time of 9 weeks for a first of its kind solution.

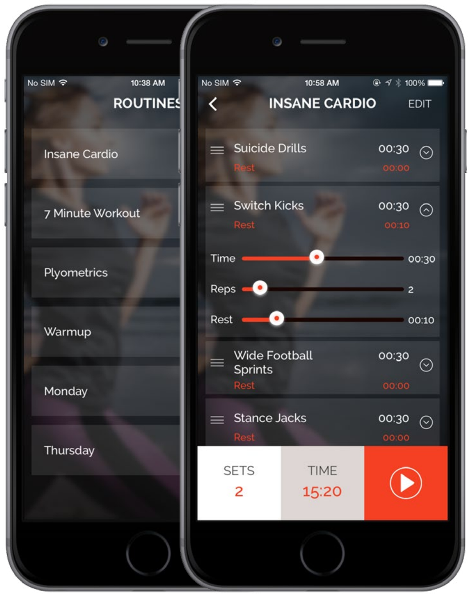
ORGANIZED CODE FOR SCALABILITY AND INCREASED RELIABILITY

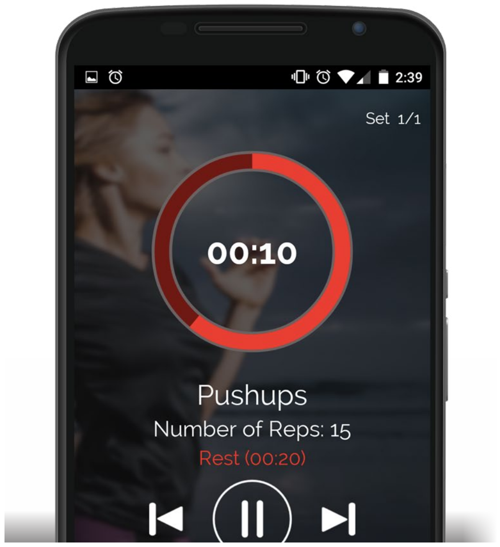
Since J2ObjC enforces Model View Controller (MVC) architecture, the Clearbridge team was able to follow these best practices for development. As a result, the foundation of the app now allows for easier implementation of additional features, improved reliability (fewer bugs) and better overall organization of code. This has been an exceptional benefit to our client as their internal team continues to iterate on the product.

CASE STUDY

HACKATHON - IDEA TO LAUNCH: 6 WEEKS, 2 PLATFORMS

A hackathon was held by Clearbridge Mobile with the goal of launching a fully functional native mobile application in 6 weeks on iOS and Android platforms. The hackathon inspired the fitness app FitCentral.





CHALLENGE

Given a hard deadline, developers were required to find innovative techniques to reduce development efforts while maintaining the highest quality of code.

CHALLENGE

Using J2ObjC, the Clearbridge team was able to house non-user interface code (data and logic) in a single library. The result was a library that handled user data, server communication and uploaded data for both iOS and Android applications. The library ultimately reduced development redundancies and additional testing time, thus allowing the team to deliver a fully functioning native product for iOS and Android within the 6 week time frame.

CONCLUSION

The use of J2ObjC in cross-platform development, while gaining some traction, is still relatively nascent. Current barriers to adoption include organizational learning, the effort required to establish new project workflows/structures, as well as defining where and when to use it.

But with the ability to reduce redundancies and shorten development cycles, it's much more cost-efficient than coding natively for each platform. Furthermore, since it enforces MVC architecture, the code is cleaner, there are fewer errors made, and the application is much easier to update when necessary. With cleaner code and reduced time and resources required, J2ObjC is a tool to seriously consider for cross-platform projects.

CONTACT

REPRESENTATIVE

Sean Huynh
Account Executive
sean@clearbridgemobile.com
647 - 361 - 8401 X 178

WEBSITE

www.clearbridgemobile.com