WILEY

MODULE [10] SESSION [4]
SESSION TITLE: [Monetize Your App]

Sources:

1. [Professional IOS programming] [Chapter 16][9781118661130]

MODULE Objectives

At the end of this module, you will be able to:

- Explain In-App Purchase and In-App Purchase helper class
- Implement advertisements in your application

Session Objectives

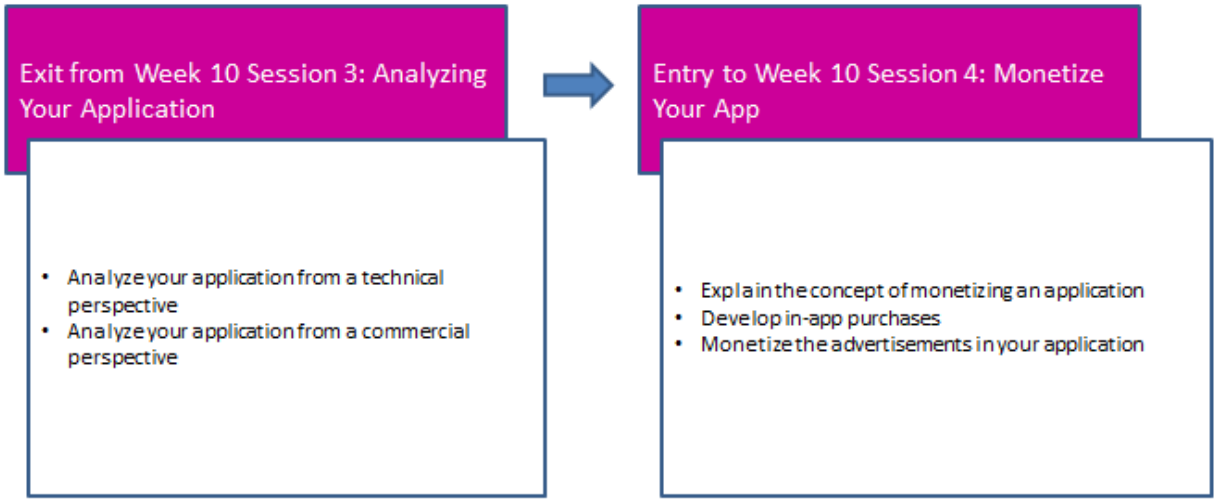
At the end of this session, you will be able to:

- Explain the concept of monetizing an application
- Develop in-app purchases
- Monetize the advertisements in your application

<H1> [Introduction]

Building an iOS application and testing it takes a lot of hard work, but all of your efforts are futile if your application is not able to earn any money. For an application to be a commercial success, it is important to monetize it well. The term monetization is used to mean the ability to generate revenue through your application.

This session takes you through the number of ways to monetize your efforts, including paid application, In-App Purchases, and advertising.



<H1> [Introduction to Monetizing]

Source: [Professional IOS programming] [Chapter 16] [447]

After all the time you have put into building and testing your application, you may want to monetize your efforts. You can do this in a number of ways, including:

- Paid application
- Advertising
- In-App Purchases
- Subscriptions
- Lead generation
- Affiliate sales

Big Picture

Once you have developed an application, it may be possible that you will launch it for free or charge some money to download the application. The most visible way to make money from an app is to charge for it. Once you have a group of people who have your app and their attention, you can find multiple ways to make money from them, including mobile advertising, In-App Purchases, and offer walls.

<H2> [Paid Application]

Source: [Professional IOS programming] [Chapter 16] [448]

Publishing your application as a paid application in the App Store is simple—you set your price tier and start marketing your application so users will buy it. Keep in mind that Apple will take 30 percent of your revenue, and customers who have bought your application might expect free updates and support for as long as they use the application. The price you can set for your application depends strongly on the functionality it offers as well as the price that is set for similar or competitive applications.

<H2> [Advertising]

Source: [Professional IOS programming] [Chapter 16] [448]

You can work with an advertising platform like Apple's iAd or Google's AdMob to generate revenue by displaying advertisements in your application. You learn how to implement these platforms in the section "*Monetizing with Advertisements*." Most advertising platforms do not pay for displaying advertisements, but you will be paid if a user taps on the advertisement and is redirected to the advertisement details.

Additional Knowhow

You can also sell space to companies that might find advertising in your application beneficial. Hundreds of advertising platforms are available, and it is up to you to decide which one works best. It might be different for each application.

<H2> [In-App Purchases]

Source: [Professional IOS programming] [Chapter 16] [448]

An In-App Purchase is a way to up-sell functionality of your application. Your application might be free, but some functionality will be unlocked when a user upgrades to a pro version using an In-App Purchase.

Various kinds of In-App Purchases are available and are explained in this session, including how you can implement them. As with paid applications, Apple also takes a 30 percent cut of your revenue.

Technical Stuff

In-App Purchase is the only mechanism by which an application can allow users to purchase content that is consumed within the application.

Quick Tip

An application cannot use a payment service like PayPal as an alternative to In-App Purchase if the item being purchased is consumed on the device.

<H2> [Subscriptions]

Source: [Professional IOS programming] [Chapter 16] [448]

Subscription-based applications include, for example, magazine applications for which a user needs to purchase a subscription to receive a new issue or content. For digital magazine applications, Apple introduced Newsstand in iOS 5.

Quick Tip

You can find more information on Newsstand at *https://developer.apple.com/newsstand*.

<H2> [Lead Generation]

Source: [Professional IOS programming] [Chapter 16] [449]

A nice example of lead generation is an application named FX, which is a currency converter used by thousands of people in more than 60 countries. You can download it from *https://itunes.apple.com/in/app/fx-convertor/id479159482?mt=8*. The commercial analysis using the Flurry Analytics framework revealed that a large number of users are living in the United Kingdom, and with additional market research revealed that quite a few of them are looking to buy a property in France to spend their retirement. Once they purchase a property in France, or any other country where they do not use the Pound as a currency, they need to convert a serious amount of Pound Sterling to Euros.

Additional Knowhow

Using the FX application, a user can request a quotation for a currency conversion. That information is shared with several professional currency conversion companies, which will pay a certain fee if that user becomes a customer.

<H2> [Affiliate Sales]

Source: [Professional IOS programming] [Chapter 16] [449]

You can generate sales for products or services by subscribing to an affiliate program like Apple's iTunes.

You can find more information about the affiliate program at *http://www.apple.com/itunes/affiliates/resources/documentation/app-store-affiliate-program.html*.

Real Life Connect

If you are using the App Store, you will find the top free applications and top paid applications. As an application developer, it is your choice to make it free or paid. The price of an application is decided by the owner of the application.

Exam Check

In your certification examination, you will be required to demonstrate your understanding of monetizing an application.

Knowledge Check 1

Q. No. 966

<H1> [Developing In-App Purchases]

Source: [Professional IOS programming] [Chapter 16] [449]

When you want to implement In-App Purchase to your application, it is important to understand the In-App Purchase process, the different types of product types and their specific behaviors and requirements, as well as how to verify the payment and supply the purchased content to the user.

<H2> [Introduction to In-App Purchase]

Source: [Professional IOS programming] [Chapter 16] [449-450]

The Store Kit framework provides you with classes that allow an application payment from a user to purchase additional content. The Store Kit framework communicates with the App Store on behalf of your application and provides localized information about the products you want to offer. You present these products to users, enabling them to purchase one or more of them. Once a user makes a selection, your application uses the Store Kit to collect payment from that user.

Basically, you can sell four different kinds of products:

- S**ubscriptions**: For example, a weekly analysis report presented in your application
- **Services**: For example, a translation of a document
- **Functionality**: For example, a pro version that unlocks functionality not available in the free version or that stops displaying advertisements
- **Content**: For example, a game level, photos, artwork, and other digital content

Quick Tip

You can find the Store Kit framework reference guide at *https://developer.apple.com/library/ios/#documentation/StoreKit/Reference/StoreKit_Collection/_index.html*.

The In-App Purchase programming guide is located at *https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction/Introduction.html*.

<H2> [Registering Products]

Source: [Professional IOS programming] [Chapter 16] [450]

Before you can sell a product using In-App Purchases, you need to register it with the App Store through iTunes Connect, which is available via the member center of the developer portal. To register a product, you need to enter a name, a description, and pricing information, as well as some other metadata that will be used by the App Store.

A product is uniquely identified by a product identifier, for which it is a common practice to use a reverse Domain Name System (DNS) name like net.yourdeveloper with your bundle identifier. For example, the product identifier for the "Upgrade to Pro" product might be **net.yourdeveloper.appname.upgradeToPro** .

<H2> [Choosing the Product Type]

Source: [Professional IOS programming][Chapter 16][450]

You can select from different product types, each with its own specific characteristics. Theses product types include:

- Consumable
- Non-consumable

- Auto-renewable subscription
- Free subscription
- Non-renewing subscription

<H3>[Consumable]

Source: [Professional IOS programming][Chapter 16][450]

A consumable In-App Purchase must be purchased each time the user needs it. As the name implies, it is consumed once and then it is no longer available.

<H3>[Non-Consumable]

Source: [Professional IOS programming][Chapter 16][450]

Non-consumable In-App Purchases must be purchased only once by users. After a non-consumable product is purchased, it is provided to all devices associated with that user's iTunes account. The Store Kit provides built-in support to restore non-consumable products on multiple devices.

<H3>[Auto-Renewable Subscription]

Source: [Professional IOS programming][Chapter 16][450]

Auto-renewable subscriptions are delivered to all the devices associated with the user's Apple ID in the same way as non-consumable products. However, auto-renewable subscriptions differ in other ways. When you create an auto-renewable subscription in iTunes Connect, you choose the duration of the subscription. The App Store automatically renews the subscription each time its term expires. If a user chooses not to renew subscription, the user's access to it is revoked after the subscription expires. Your application is responsible for validating whether a subscription is currently active and can also receive an updated receipt for the most recent transaction.

<H3>[Free Subscription]

Source: [Professional IOS programming][Chapter 16][451]

Free subscriptions are a way for you to put free subscription content in Newsstand. Once a user signs up for a free subscription, the content is available on all devices associated with the user's Apple ID. Free subscriptions do not expire and can be offered only in Newsstand-enabled apps.

<H3>[Non-Renewing Subscription]

Source: [Professional IOS programming][Chapter 16][451]

A non-renewing subscription is a mechanism for creating products with a limited duration. Non-renewing subscriptions differ from auto-renewable subscriptions in the following key ways:

- The term of the subscription is not declared when you create a product in iTunes Connect. Your application is responsible for providing this information to the user. In most cases, you would include the term of the subscription in the description of your product.
- Non-renewing subscriptions can be purchased multiple times (like a consumable product) and are not automatically renewed by the App Store. You are responsible for implementing the renewal process inside your application. Specifically, your application must recognize when the subscription has expired and prompt the user to purchase the product again.
- You are required to deliver non-renewing subscriptions to all devices owned by the user. Non-renewing subscriptions are not automatically synchronized to all devices by Store Kit. You must implement this infrastructure yourself. For example, most subscriptions are provided by an external server. Your server would need to implement a mechanism to identify users and associate subscription purchases with the user who purchased them.

<H2> [Understanding the In-App Purchase Process]

Source: [Professional IOS programming][Chapter 16][451-452]

The Store Kit framework does not cover all the steps in an In-App Purchase. Its main functionality is to interact with the App Store.

The In-App Purchase process requires you to implement the following steps:

1. Retrieve a list of product identifiers that are available from the application bundle.
2. Send a request to the App Store using the Store Kit framework to retrieve detailed information on each product identifier, such as name, description, price, and product type.
3. The App Store returns the requested information to the application.
4. The application presents a user interface with the available products like a normal e-commerce store, where the user can select one or more products to purchase.
5. Once the user initiates the actual purchase, the application sends a payment request to the App Store using the Store Kit framework.
6. The App Store processes the payment and returns the transaction information to the application.

7. The application processes the received payment information and, if payment has been successful, it delivers the product to the application.

With this process your product identifiers are part of the application bundle. Although this process is easy to implement, it is difficult to maintain because if you want to add new products available for your application, you need to add those new products to your application bundle and upload a new version to the App Store. This will then go through the normal review process.

As an alternative to having to upload a new version of your application to be reviewed and released, Apple advises to store your list of product identifiers on a server. In the first step of the In-App Purchase process, you contact your server to obtain the list of product identifiers. In this scenario, you can add new products without having to upload a new version of your application to be reviewed and released.

<H2> [Implementing an In-App Purchase]

Source: [Professional IOS programming][Chapter 16][452-453]

In this section, you will learn how to implement a complete In-App Purchase process for an application that has several product identifiers available from the application bundle.

Start Xcode and create a new project using the **Single View Application** template (see **Figure 1**). Name it **MyStore** using the options as shown in **Figure 2**.

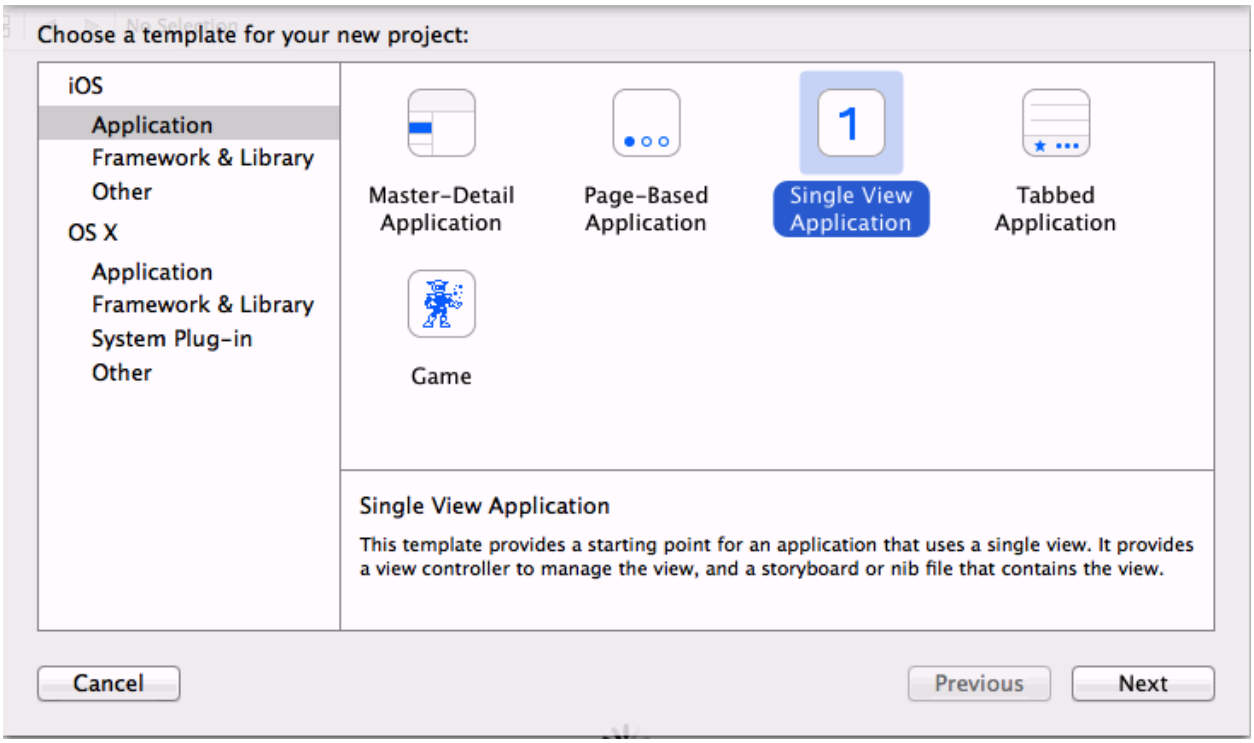


Figure 1: Select the Single View Application Template

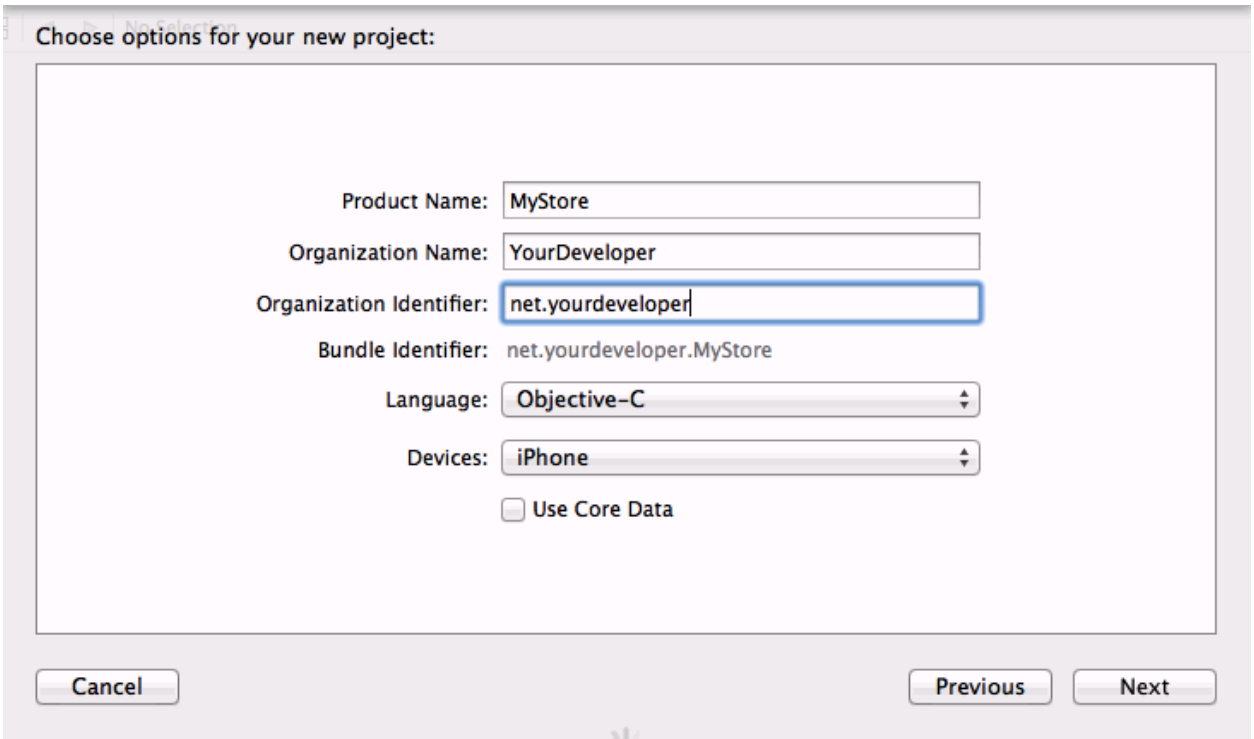
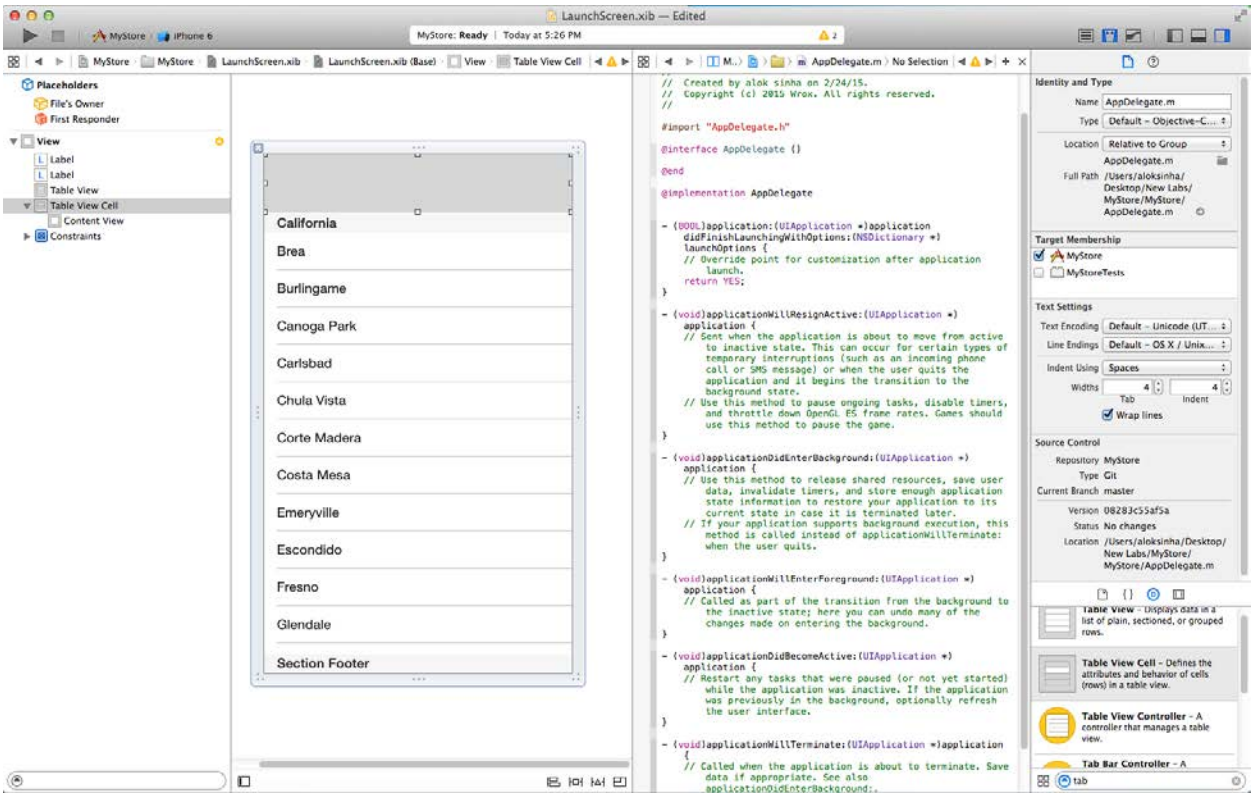


Figure 2: Create a New Project, MyStore

Add the Store Kit framework to your project. In this application, you will be presenting a store with several products.

Let's assume that your application has an option to purchase a pro version that does not present any advertisements, which the standard application does. Later in this session, in the section "*Monetizing with Advertisements*," you will learn how to implement the advertisements, but for now you will just display a surrogate image as an advertisement.

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface with a `UITableView`. Note that the `UITableView` has a Y position of 50 so you can display the `surrogateAdvertisementView` above it. This `surrogateAdvertisementView` will later be replaced with a real advertisement.



<H3>[Create Products in iTunes Connect]

Source: [Professional IOS programming][Chapter 16][453-459]

You have to start by creating your products in iTunes Connect. The *https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html* link will bring you to the iTunes Connect Developer Guide, which explains in detail how to create and configure your products.

For this example project, you will configure three products in iTunes Connect, as shown in **Table 1**.

TABLE 1: Sample Products in iTunes Connect

Product Identifier	Product Type

net.yourdeveloper.MyStore.cons1	Consumable
net.yourdeveloper.MyStore.cons2	Consumable
net.yourdeveloper.MyStore.upgrade	Non-Consumable

To make life easier, create an In-App Purchase helper class named `YDInAppPurchaseManager`. You can add the In-App Purchase helper class to the Personal Library and reuse it in all your applications for which you want to use the In-App Purchase functionality.

Using Xcode, select **File ➪ New ➪ File** from the menu. Select the **Objective-C File** template and click **Next**, as shown in **Figure 3**.

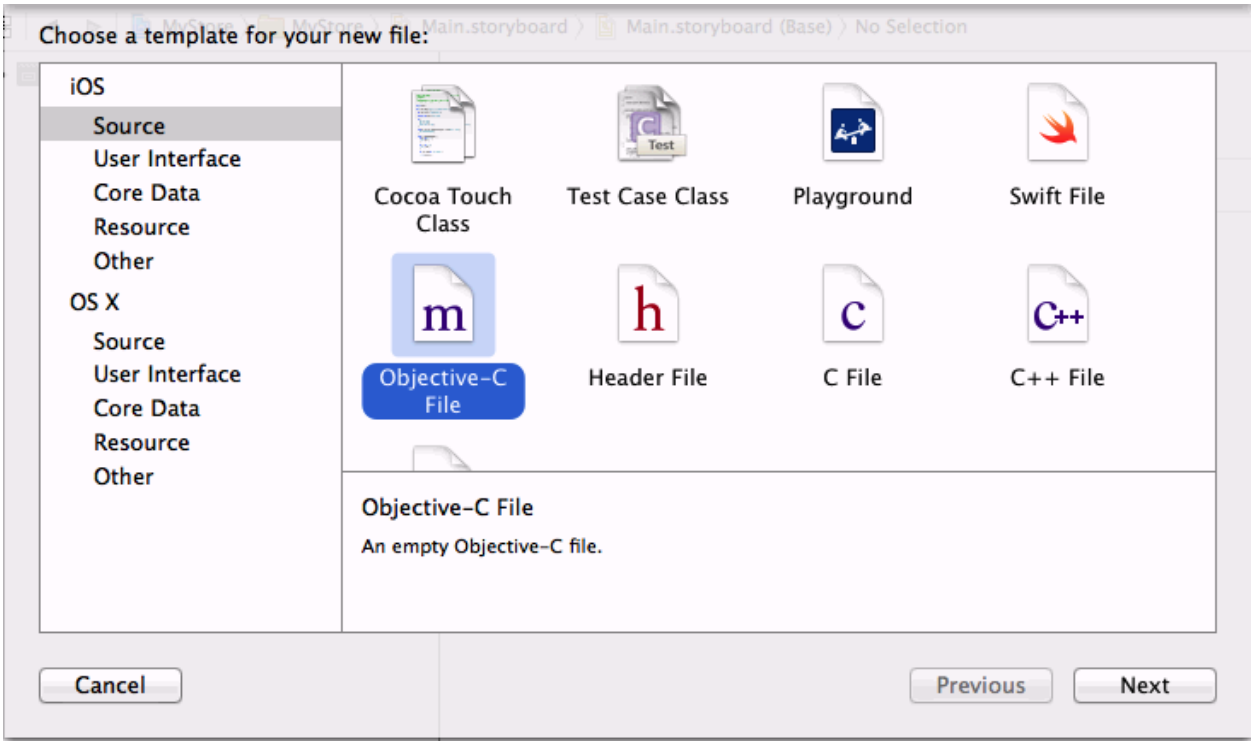


Figure 3: Select the Objective-C File Template

In the **Choose options for your new file** dialog box, name the class as `YDInAppPurchaseManager` and subclass it from `NSObject`, as shown in **Figure 4**.

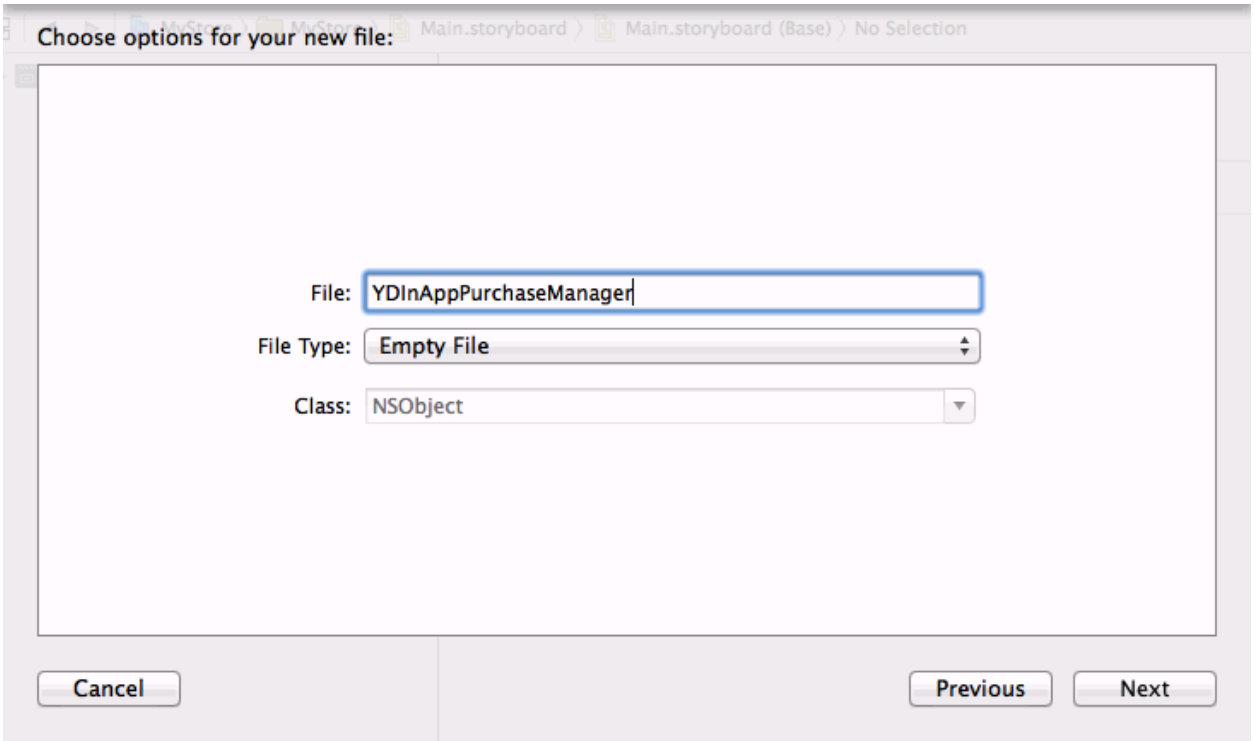


Figure 4: Create the `YDInAppPurchaseManager` Class in Objective-C

The `YDInAppPurchaseManager` class will be designed as a singleton class. Singleton classes are an important concept to understand because they exhibit an extremely useful design pattern. This idea is used throughout the iPhone SDK. For example, `UIApplication` has a method called `sharedApplication` that, when called from anywhere, returns the `UIApplication` instance that relates to the currently running application.

In the `YDInAppPurchaseManager.h` file, create the `sharedInstance` method as follows:

```
+ (id)sharedInstance;
```

Implement this method in the `YDInAppPurchaseManager.m` file. This method starts by creating a static variable named `sharedManager`, which is then initialized only once. The only-once creation is realized by using the `dispatch_once` function from Grand Central Dispatch, as shown below:

```
+ (id)sharedInstance {
      static YDInAppPurchaseManager *sharedManager = nil;
```

```
      static dispatch_once_t onceToken;
      dispatch_once(&onceToken, ^{
            sharedManager = [[self alloc] init];
      });
      return sharedManager;
}
```

You can now call the methods of the `YDInAppPurchaseManager` by calling `[[YDInAppPurchaseManager sharedInstance] METHOD_NAME];` where you replace `METHOD_NAME` with one of the implemented methods.

Open the `YDInAppPurchaseManager`, import the `StoreKit` header file, and define some constants, which will be used for setting up notifications.

Create two strong `NSArray` properties:

- `availableProducts` - This property will contain the available products sellable via My Store application.
- `invalidProducts` - This property will contain products that are not configured properly or are not available in the App Store.

Next, create the definitions for the public methods. The complete `YDInAppPurchaseManager.h` file is shown below:

```
#import <Foundation/Foundation.h>

#import <StoreKit/StoreKit.h>

//Purchase
//Public
#define kYDInAppPurchaseManagerProductsFetchedNotification
      @"kInAppPurchaseManagerProductsFetchedNotification"
#define kYDInAppPurchaseManagerTransactionFailedNotification
      @"kInAppPurchaseManagerTransactionFailedNotification"
#define kYDInAppPurchaseManagerTransactionSucceededNotification
      @"kInAppPurchaseManagerTransactionSucceededNotification"
#define kYDInAppPurchaseManagerProductListRetrievedNotification
```

```
      @"kYDInAppPurchaseManagerProductListRetrievedNotification"
#define kYDInAppPurchaseManagerDeliverProduct
      @"kYDInAppPurchaseManagerDeliverProduct"

@interface YDInAppPurchaseManager : NSObject
//This class is a singleton class
+ (id)sharedInstance;

@property(nonatomic,strong)NSArray* availableProducts;
@property(nonatomic,strong)NSArray* invalidProducts;

// public methods
- (void)retrieveProductListFromURL:(NSURL *)urlToJsonFile
            applicationName:(NSString *)appName;
- (BOOL)productPurchased:(NSString *)productIdentifier;
- (BOOL)canMakePurchases;
- (void)purchaseProduct:(SKProduct*)productToPurchase;
@end
```

Add the `URLRequest` class you created in the previous session to your project. For the validation of the transaction receipt, you need the ability to create a base64-encoded string from an `NSData` object.

In Xcode, select File ➪ New ➪ File from the menu and select the **Objective-C File** template. On the next screen, name the file as **base64**, file type as **Category**, and class as **`NSString`,** as shown in **Figure 5**.

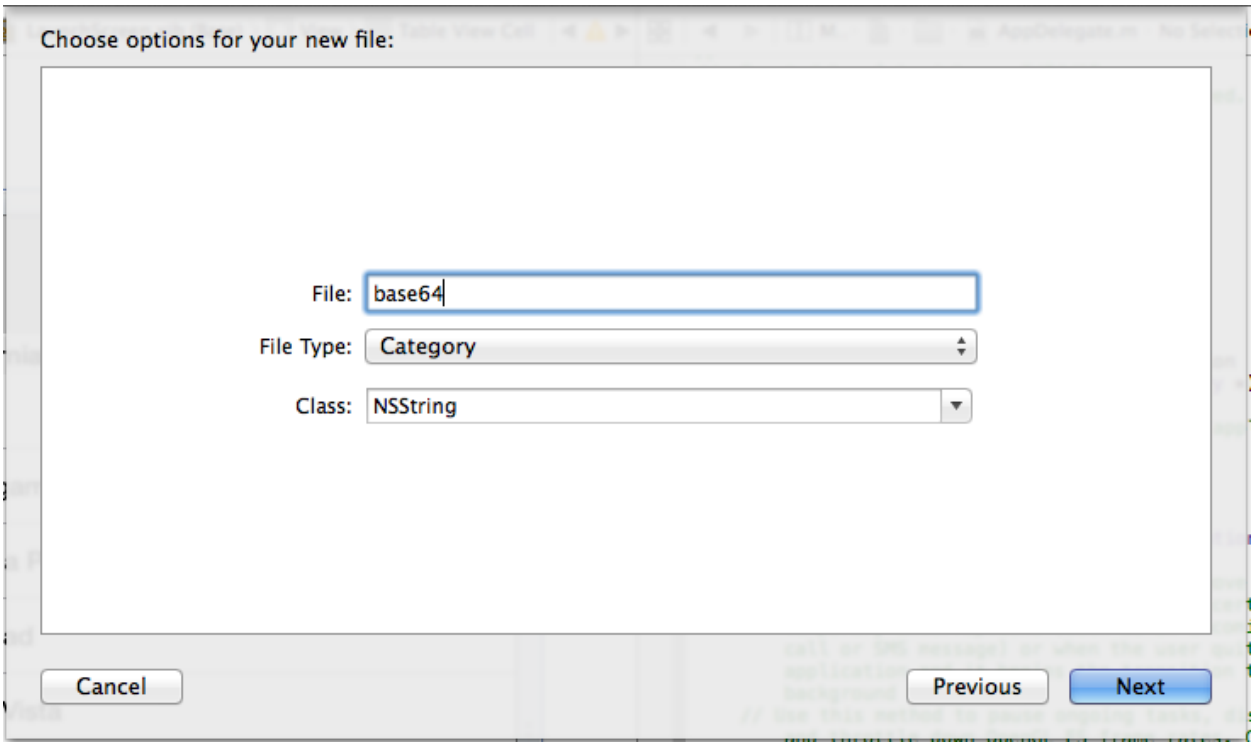


Figure 5: Choose Options for base64

Open the `NSString + base64.h` file and implement the code as shown below:

```
#import <Foundation/Foundation.h>

@interface NSString (base64)
+ (NSString *) base64StringFromData:(NSData *)paramData
length:(NSInteger)paramLength;
@end
```

Open the `NSString+base64.m` file and implement the code as shown below:

```
#import "NSString+base64.h"
@implementation NSString (base64)
static char Base64EncodingTable[64] = {
      'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
      'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
```

```
        'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
        'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'
};
+ (NSString *) base64StringFromData:(NSData *)paramData length:(NSInteger)paramLength
{


        NSInteger ixtext = 0, dataLength = 0;
        long remainingCharacters;
        unsigned char inputCharacters[3] = {0, 0, 0}, outputCharacters[4] = {0, 0, 0,
        0};
        short counter, charsonline = 0, charactersToCopy = 0;
        const unsigned char *rawBytes;


        NSMutableString *result;


        dataLength = [paramData length];
        if (dataLength < 1){
                return [NSString string];
        }
        result = [NSMutableString stringWithCapacity: dataLength];
        rawBytes = [paramData bytes];
        ixtext = 0;


        while (YES) {
                remainingCharacters = dataLength - ixtext;
                if (remainingCharacters <= 0)
                        break;
                for (counter = 0; counter < 3; counter++) {
                        NSInteger index = ixtext + counter;
                        if (index < dataLength)
                                inputCharacters[counter] = rawBytes[index];
                        else
                                inputCharacters[counter] = 0;
                }
                outputCharacters[0] = (inputCharacters[0] & 0xFC) >> 2;


                outputCharacters[1] =
                   ((inputCharacters[0] & 0x03) << 4) | ((inputCharacters[1] & 0xF0) >>
                4);
                outputCharacters[2] =
```

```
            ((inputCharacters[1] & 0x0F) << 2) | ((inputCharacters[2] & 0xC0) >>
        6);
        outputCharacters[3] = inputCharacters[2] & 0x3F;
        charactersToCopy = 4;
        switch (remainingCharacters) {
        case 1:
           charactersToCopy = 2;
            break;
        case 2:
            charactersToCopy = 3;
            break;
        }
        for (counter = 0; counter < charactersToCopy; counter++){
               [result appendString: [NSString stringWithFormat:
                 @"%c", Base64EncodingTable[outputCharacters[counter]]]];
        }
        for (counter = charactersToCopy; counter < 4; counter++){
               [result appendString: @"="];
        }
        ixtext += 3;
        charsonline += 4;

        if ((paramLength > 0) && (charsonline >= paramLength)){
               charsonline = 0;
        }
    }
    return result;
}
@end
```

Open the `YDInAppPurchaseManager.m` file and import the `URLRequest` and the `NSString+base64` header files.

To validate the transaction receipts, you need to access a URL on the apple.com domain. There are different URLs for the production and the sandbox environments. For easy access, just create them here as constants as shown below:

```
const NSString* kSKTransactionReceiptVerifierURL =
    @"https://buy.itunes.apple.com/verifyReceipt";
const NSString* kSKSandboxTransctionReceiptVerifierURL =
```

```
      @"https://sandbox.itunes.apple.com/verifyReceipt";
```

Subscribe to the `SKProductRequestDelegate` and `SKPaymentTransactionObserver` protocols. Create a private variable named `productRequest` of type `SKProductRequest` and one named `productList` of type `NSArray` in the private interface declaration, as shown below:

```
@interface YDInAppPurchaseManager() <SKProductsRequestDelegate,
                                     SKPaymentTransactionObserver>
{
      SKProductsRequest *productsRequest;
      NSMutableArray* productList;
}
@end
```

Create the `sharedInstance` method as explained earlier and shown below:

```
+ (id)sharedInstance {
   static YDInAppPurchaseManager *sharedManager = nil;
   static dispatch_once_t onceToken;
   dispatch_once(&onceToken, ^{
      sharedManager = [[self alloc] init];
   });
   return sharedManager;
}
```

<H3>[Retrieving the Product List]

Source: [Professional IOS programming][Chapter 16][459-461]

As you have seen before, you can retrieve the product list in two ways: you can create a list of product identifiers in your application bundle or you can store the list on a server.

When integrating the list in the bundle, you need to build and upload a new version of your application once the product list changes. However, when you store it on a server, you are free to change the product list whenever you want. Because it does not affect the application, there is no need to build a new version and submit it to the App Store.

You can develop a complete backend system in which you manage your application and its In-App Purchase product lists, or you can use a text editor and create a JSON file with a structure similar to that shown below:

```
{
  "applications": {
   "application": [
   {

      "name": "MyStore",
      "products": {
       "product": [
      {
       "product_identifier": "net.yourdeveloper.MyStore.cons1",
       "product_name": "Consumable 1",
       "product_type": "Consumable"
      },
      {
       "product_identifier": "net.yourdeveloper.MyStore.cons2",
       "product_name": "Consumable 2",
       "product_type": "Consumable"
      },
      {
       "product_identifier": "net.yourdeveloper.MyStore.upgrade",
       "product_name": "Upgrade to pro",
       "product_type": "Non-Consumable"
      }
      ]
     }
    },
{
"name": "FX",
 "products": {
   "product": {
      "product_identifier": "net.yourdeveloper.FX.upgrade",
      "product_name": "Upgrade to Pro",
      "product_type": "Non consumable"
      }
```

```
    }
   }
  ]
 }
}
```

With a structure like this, you can support multiple applications with multiple products using a single JSON file. You can create it in any text editor and it does not require a back end. The only requirement is that it is accessible via a public URL and the web server is configured for the MIME type JSON to serve the contents of the file to the request.

Implement the `retrieveProductListFromURL:applicationName:` method. It starts by initializing the local `productList`, and creates and initializes the request to retrieve the information from the passed URL. The received data, which is the JSON response, is serialized into an `NSDictionary`. The `NSDictionary` is parsed, and the products that match the `applicationName` you have passed are added to the `productList` array. When all entries from the `NSDictionary` are processed, the `requestProductDetails` method is called. The method implementation is shown below:

```
-(void)retrieveProductListFromURL:(NSURL *)urlToJsonFile
      applicationName:(NSString *)appName;
{
      if (productList)
       productList=nil;
      productList=[[NSMutableArray alloc] init];
      NSMutableURLRequest        *request        =        [NSMutableURLRequest
      requestWithURL:urlToJsonFile
                        cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
                        timeoutInterval:60];
[request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
[urlRequest  startWithCompletion:^(URLRequest  *request,  NSData  *data,  BOOL
success) {
      if (success)
            {
            NSError* error=nil;
            NSDictionary* resultDict =
                  [NSJSONSerialization JSONObjectWithData:data
```

```
                options:kNilOptions error:&error];
            NSDictionary* applications =
                (NSDictionary *)[resultDict objectForKey:@"applications"];
            for (NSDictionary* applicationDict in (NSDictionary* )
                [applications objectForKey:@"application"])
            {
                NSString*       thisAppName       =       [applicationDict
                objectForKey:@"name"];
                //check if this is the appname we are looking for
                if ([thisAppName isEqualToString:appName])
            {
                        NSDictionary* appProducts =
                        (NSDictionary*)[applicationDict
                        objectForKey:@"products"];
                        for (NSDictionary* product in (NSDictionary*)
                         [appProducts objectForKey:@"product"])
                        {
                        NSString* productIdentifier =
                              [product objectForKey:@"product_identifier"];
                              [productList addObject:productIdentifier];
                        }
                }
            }
                [self requestProductDetails];
            }
            else
            {
            NSLog(@"error %@", [[NSString alloc] initWithData:data
                encoding:NSUTF8StringEncoding]);
            }
      }];
}
```

Quick Tip

The JSON file is available via the URL *http://developer.yourdeveloper.net/products.json*. You are free to download it and make the modifications required for your own applications and products and store it on your own server.

<H3>[Requesting Product Detail Information]

Source: [Professional IOS programming][Chapter 16][462]

After you have created the product list containing your product identifiers, create and initialize your SKProductRequest instance variable named productsRequest using the initWithProduct Identifers method of the SKProductRequest class. Set the delegate to self and call the start method of the SKProductRequest class.

The SKProductRequest class is used to retrieve localized information about the list of products you have passed in the initWithProductIdentifiers: method. When the request completes, the productRequest:didReceiveReponse: delegate method is called, which you will implement next. The requestProductDetails method is shown below:

```
- (void)requestProductDetails
{
      NSSet *productIdentifiers = [NSSet setWithArray:productList];
      productsRequest = [[SKProductsRequest alloc]
                         initWithProductIdentifiers:productIdentifiers];
      productsRequest.delegate = self;
      [productsRequest start];
}
```

<H3>[Receiving Product Detail Information]

Source: [Professional IOS programming][Chapter 16][462-463]

Implement the productRequest:didReceiveReponse: delegate method of the SKProductRequest class. This delegate method is called when the SKProductRequest start method is finished, and it will return with a response of type SKProductResponse.

The SKProductResponse class has two public read-only properties of type NSArray:

- The products array, which contains the products that are available for sale
- Invalid product Identifiers, which contains product identifiers that are invalid. A product identifier might be invalid for the following reasons:
    - ➢ Not released for sale in iTunes Connect
    - ➢ Not complete or correctly configured

You assign the two response arrays to your YDInAppPurchaseManager properties, and post a notification named kYDInAppPurchaseManagerProductsFetchedNotification to inform the observers that the product details are available and can be displayed in the user interface. The products Request:didReceiveResponse: method is shown below:

```
- (void)productsRequest:(SKProductsRequest *)request
      didReceiveResponse:(SKProductsResponse *)response
{
      self.availableProducts = response.products;
      self.invalidProducts= response.invalidProductIdentifiers;
      //notify the products have been fetched
      [[NSNotificationCenter defaultCenter]
      postNotificationName:kYDInAppPurchaseManagerProductsFetchedNotification
      object:self userInfo:nil];
}
```

<H3>[Present the Store]

Source: [Professional IOS programming][Chapter 16][463]

Open the YDViewController.h file and import the YDInAppPurchaseManager header file. Create a strong property of type UIView named surrogateAdvertisementView, which will be used to simulate an advertisement that will be replaced with a real advertisement in the section "*Monetizing with Advertisements*." The YDViewController.h implementation is shown below:

```
#import <UIKit/UIKit.h>

#import "YDInAppPurchaseManager.h"
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property(nonatomic,strong) UIView* surrogateAdvertisementView;
@end
```

Open the YDViewController.m file and subscribe to the UITableViewDataSource and UITableViewDelegate protocols. In the viewDidLoad method, create the surrogateAdvertisementView as a simple UIView with a blue background to simulate an

advertisement at the top of the screen. Finally, call the `initializeInAppPurchaseManager` method in the `viewDidLoad` method.

<H3>[Initialize the YDInAppPurchaseManager]

Source: [Professional IOS programming][Chapter 16][463-464]

Create the method `initializeInAppPurchaseManager`, which will start by calling the `canMakePurchases` method of the `YDInAppPurchaseManager` class to see if the application is capable of making a purchase. Set up the observers to process the notifications posted by the `YDInAppPurchaseManager` and create an NSURL object with the URL for your JSON product list file. Finally, call the `retrieveProductListFromURL:applicationName:` method of the `YDInAppPurchaseManager` class.

The implementation is shown below:

```
-(void)initializeInAppPurchaseManager
{
      if ([[YDInAppPurchaseManager sharedInstance] canMakePurchases])
      {


            //add notification called when products have been fetched
            [[NSNotificationCenter defaultCenter] addObserver:self
                  selector:@selector(productListReceived:)
                  name:kYDInAppPurchaseManagerProductsFetchedNotification
                  object:nil];

            [[NSNotificationCenter defaultCenter] addObserver:self
                  selector:@selector(productPurchaseSucceeded:)
                  name:kYDInAppPurchaseManagerTransactionSucceededNotificatio
                  n
                  object:nil];

            NSURL* url = [NSURL
```

```
            URLWithString:@"http://developer.yourdeveloper.net/products.json"
            ];
            [[YDInAppPurchaseManager sharedInstance]
            retrieveProductListFromURL:url applicationName:@"MyStore"];
      }
      else
      {
            UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error"
                        message:@"you can't make purchases" delegate:self
                        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];
            [alert show];
      }
}
```

The `productsLoaded:` method is called once the `kYDInAppPurchaseManagerProductsFetchedNotification` notification is posted to inform you that the products have been loaded. You can simply call the `reloadData` method of your `UITableView` instance as shown below:

```
- (void)productListReceived:(NSNotification *)notification
{
      [self.mTableView reloadData];
}
```

<H3>[Display the Available Products]

Source: [Professional IOS programming][Chapter 16][464-467]

To display the available products in your `UITableView` instance, you need to implement, at minimum, the mandatory delegate methods. The `numberOfSections InTableView:method` returns 1 because your `UITableView` will have only one section. The `table view:numberOfRowsInSection:` method will return the value of the `availableProducts` property of the `YDInAppPurchaseManager` class. This property has been set by the `productRequest:didReceiveResponse:` method, as shown in the following code:

```
-(void)productsRequest:(SKProductsRequest *)request
      didReceiveResponse:(SKProductsResponse *)response
{
```

```
      self.availableProducts = response.products;
      self.invalidProducts= response.invalidProductIdentifiers;
      //notify the Products have been fetched
      [[NSNotificationCenter defaultCenter]
      postNotificationName:kYDInAppPurchaseManagerProductsFetchedNotificatioo
bject:self useInfo:nil]
}
```

To display each available product with a localized description and a localized price, implement the `tableView:cellForRowAtIndexPath:` method as follows:

1. Create a reusable `UITableViewCell`.
2. Get the `SKProduct` from the `availableProducts` array of the `YDInAppPurchaseManager` class for the `indexPath.row` index.
3. Set the `LocalizedTitle` property of the retrieved `SKProduct` to the `cell.textLabel.text` property.
4. Create, initialize, and configure an `NSNumberFormatter` named `priceFormatter`.
5. Set the formatted localized price to the `cell.detailTextLabel.text` property.
6. Perform a check to see if the product has been purchased already by calling the `productPurchased:` method of the `YDInAppPurchaseManager` class by passing the `productIdentifier` property of the selected `SKProduct`.
7. Display a checkmark if the product is already purchased. If not, then create a `UIButton` with the title Buy and set the `indexPath.row` value to its tag property.
8. Finally, return the cell object.

The three `UITableView` delegate methods are shown below:

```
- (void)productListReceived:(NSNotification *)notification
{
      [self.mTableView reloadData];
}

#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
      return 1;
}
-  (NSInteger)  tableView:  (UITableView  *)  table  numberOfRowsInSection:
(NSInteger)section
{
```

```
return [[[YDInAppPurchaseManager sharedInstance] availableProducts] count];
}
- (UITableViewCell *) tableView: (UITableView *) tableView
cellForRowAtIndexPath: (NSIndexPath *) indexPath
{
      UITableViewCell *cell = (UITableViewCell *)[tableView
      dequeueReusableCellWithIdentifier:@"MyCellIdentifier"];
      if (cell == nil) {
            cell            =            [[UITableViewCell            alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:@"MyCellIdentifier"];
      }
      cell.selectionStyle = UITableViewCellSelectionStyleNone;
      SKProduct*   product   =   [[[YDInAppPurchaseManager   sharedInstance]
      availableProducts]
                                  objectAtIndex:indexPath.row];
      cell.textLabel.text = product.localizedTitle ;
      // Create an NSNumberFormatter
      NSNumberFormatter* priceFormatter;


      priceFormatter = [[NSNumberFormatter alloc] init];
      [priceFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
      [priceFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
      [priceFormatter setLocale:product.priceLocale];


      cell.detailTextLabel.text               =               [priceFormatter
      stringFromNumber:product.price];


      if ([[YDInAppPurchaseManager sharedInstance]
            productPurchased:product.productIdentifier]) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
        cell.accessoryView = nil;
      } else {
      UIButton*              buyButton              =              [UIButton
      buttonWithType:UIButtonTypeRoundedRect];
      buyButton.frame = CGRectMake(0, 0, 72, 37);
      [buyButton setTitle:@"Buy" forState:UIControlStateNormal];
      buyButton.tag = indexPath.row;
```

```
        [buyButton addTarget:self action:@selector(buyButtonTapped:)
              forControlEvents:UIControlEventTouchUpInside];
        cell.accessoryType = UITableViewCellAccessoryNone;
        cell.accessoryView = buyButton;
        }

        return cell;
}
```

The result is a user interface that will look similar to the one shown in **Figure 6**.

Figure 6: User Interface Result

<H3>[Buy a Product]

Source: [Professional IOS programming][Chapter 16][467]

Now that your user interface displays a table with available products and a **Buy** button for each product, you need to implement the `buyButtonTapped:` method, which is invoked if the user taps one of the visible **Buy** buttons. This method creates a `UIButton` named `buyButton` by casting the sender value.

Next, it creates a `SKProduct` instance named product from the `availableProducts` array of the `YDInAppPurchaseManager` class by using the `objectAtIndex:` method. It then calls the `purchase-Product:` method of the `YDInAppPurchaseManager` class, as shown in the following code:

```
- (void)buyButtonTapped:(id)sender
{
      UIButton* buyButton = (UIButton *)sender;
      SKProduct*   product   =   [[[YDInAppPurchaseManager   sharedInstance]
      availableProducts]
                                    objectAtIndex:buyButton.tag];
      [[YDInAppPurchaseManager sharedInstance] purchaseProduct:product];
}
}
```

<H3>[Sending a Payment Request]

Source: [Professional IOS programming][Chapter 16][468-469]

Open the `YDInAppPurchaseManager` and create the `purchaseProduct:` method. This method calls the `defaultQueue addTransactionObserver:` method of the default `SKPaymentQueue`. It creates and initializes a variable named payment of type `SKPayment` by calling the `paymentWithProduct:` method of the `SKPayment` class. The payment instance is added to the default `SKPayment` queue by calling the `addPayment:` method, as shown below:

```
- (void)purchaseProduct:(SKProduct*)productToPurchase
{
      [[SKPaymentQueue defaultQueue] addTransactionObserver:self];
      SKPayment* payment = [SKPayment paymentWithProduct:productToPurchase];
      [[SKPaymentQueue defaultQueue] addPayment:payment];
```

```
}
```

You have added a `TransactionObserver` for the `SKPaymentQueue`, and now you implement the observer itself by implementing the `paymentQueue:updatedTransaction:` method as shown in the following code. Based on the transaction state, one of the following methods is called:

- **`completeTransaction`**: This method is called if the transaction has been completed.
- **`failedTransaction`**: This method is called if the transaction fails or if the user taps **Cancel**.
- **`restoreTransaction`**: This method is called if the transaction needs to be restored.

```
#pragma mark SKPaymentTransactionObserver methods
// called when the transaction status is updated
-   (void)paymentQueue:(SKPaymentQueue   *)queue   updatedTransactions:(NSArray
*)transactions
{
      for (SKPaymentTransaction *transaction in transactions)
            {
                  switch (transaction.transactionState)
                  {
                        case SKPaymentTransactionStatePurchased:
                        [self completeTransaction:transaction];
                        break;
                        case SKPaymentTransactionStateFailed:
                        [self failedTransaction:transaction];
                        break;
                        case SKPaymentTransactionStateRestored:
                        [self restoreTransaction:transaction];
                        break;
                        default:
                        break;
                  }
            }
}
- (void)completeTransaction:(SKPaymentTransaction *)transaction
{
[self finishTransaction:transaction wasSuccessful:YES];
}
- (void)restoreTransaction:(SKPaymentTransaction *)transaction
{
```

```
[self finishTransaction:transaction wasSuccessful:YES];
}
- (void)failedTransaction:(SKPaymentTransaction *)transaction
{
if (transaction.error.code != SKErrorPaymentCancelled)
{
// error!
[self finishTransaction:transaction wasSuccessful:NO];
}
else
{
// this is fine, the user just cancelled, so don't notify
[[SKPaymentQueue defaultQueue] finishTransaction:transaction];
}
}
```

Both the `completeTransaction:` method and the `restoreTransaction:` method call the `finishTransaction:wasSuccessful` method, which you will implement next.

<H3>[Processing the Transaction Receipt]

Source: [Professional IOS programming][Chapter 16][469]

The `finishTransaction:wasSuccessful` method is called once the payment transaction has finalized. If the result of the transaction was not successful, a `kYDInAppPurchaseManagerTransactionFailedNotification` notification is posted, for which you have set up an observer in the `YDViewController.m` file and implemented a method to handle the response.

You start by calling the `finishTransaction:` method on the default `SKPaymentQueue`.

<H3>[Verifying the Transaction Receipt]

Source: [Professional IOS programming][Chapter 16][469-471]

Your application should now perform the additional steps to verify that the payment you received from Store Kit came from Apple. To verify the receipt, implement the following steps:

1. Retrieve the receipt data from the `transactionReceipt` property and encode it as a `base64` string using the `NSString+base64` category.
2. Create a JSON object with a single key named receipt-data and the base64-encoded string as the value.
3. Post the JSON object using an HTTP POST request to the verification URL using the `URLRequest` class.
4. The received response, which is again a JSON object with two keys, status and receipt, should have the value 0 for the status key in case of a valid receipt. Any value other than 0 means the receipt is invalid.
5. Once you have verified the receipt is valid, post a notification named `kYDInAppPurchaseManagerTransactionSucceededNotification` to inform the observer that the payment has succeeded and has been verified.
6. Finally, post a notification named `kYDInAppPurchaseManagerDeliverProductNotification` to inform the observer that the payment has succeeded and has been verified.

Note there are two URLs - *https://buy.itunes.apple.com/verifyReceipt* for production and *https://sandbox.itunes.apple.com/verifyReceipt* for the sandbox environment.

The `finishTransaction:wasSuccessful:` method is shown in the following code:

```
- (void)finishTransaction:(SKPaymentTransaction *)transaction
                      wasSuccessful:(BOOL)wasSuccessful
{
      // remove the transaction from the payment queue.
      [[SKPaymentQueue defaultQueue] finishTransaction:transaction];

      NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                      transaction, @"transaction" , nil];
      if (wasSuccessful)
      {
      //Verify the transaction receipt if it really came from Apple
      NSError *jsonSerializationError = nil;

      NSString *transactionReceiptAsString = [NSString
            base64StringFromData:transaction.transactionReceipt
            length:[transaction.transactionReceipt length]];
```

```
      NSDictionary *receiptDictionary = [[NSDictionary alloc]
            initWithObjectsAndKeys:transactionReceiptAsString,
            @"receipt-data", nil];
      id receiptDictionaryAsData = [NSJSONSerialization
                              dataWithJSONObject:receiptDictionary
                              options:NSJSONWritingPrettyPrinted
                              error:&jsonSerializationError];


      NSURL *sandboxStoreURL = [[NSURL alloc]
                  initWithString:kSKSandboxTransctionReceiptVerifierURL];
      NSURL      *storeURL      =      [[NSURL      alloc]      initWithString:
      kSKTransactionReceiptVerifierURL];
      NSMutableURLRequest        *request        =        [NSMutableURLRequest
      requestWithURL:sandboxStoreURL
      cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
      timeoutInterval:60];
      [request setHTTPMethod:@"POST"];
      [request setHTTPBody:receiptDictionaryAsData];
      [request    setValue:@"application/json"    forHTTPHeaderField:@"Content-
      Type"];
      URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
   [urlRequest startWithCompletion:^(URLRequest *request, NSData *data, BOOL
success) {
if (success)
      {
      NSError* error=nil;
      NSDictionary* resultDict = [NSJSONSerialization JSONObjectWithData:
                  data options:kNilOptions error:&error];
      [[NSUserDefaults standardUserDefaults] setObject:resultDict
                  forKey:transaction.payment.productIdentifier];
      [[NSUserDefaults standardUserDefaults] synchronize];
      }
else
      {
      NSLog(@"error %@", [[NSString alloc] initWithData:data
                              encoding:NSUTF8StringEncoding]);
      }
}];
      [[NSNotificationCenter defaultCenter]
```

```
      postNotificationName:kYDInAppPurchaseManagerTransactionSucceededNotific
      ation
      object:self userInfo:userInfo];
[[NSNotificationCenter defaultCenter]
      postNotificationName:kYDInAppPurchaseManagerDeliverProductNotification
      object:self userInfo:userInfo];
}
else
{
      // send out a notification for the failed transaction
      [[NSNotificationCenter defaultCenter]
      postNotificationName:kYDInAppPurchaseManagerTransactionFailedNotificati
      on
      object:self userInfo:userInfo];
      }
}
```

<H3>[Delivering the Product]

Source: [Professional IOS programming][Chapter 16][471-473]

The notification observer you configured will be invoked by the `finishTransaction:wasSuccessful:` method. As a last step, you need to open the `YDViewController.m` file and implement the following four methods to handle the received notifications:

- `productPurchaseSucceeded:`
- `productPurchaseFailed:`
- `deliverProduct:`
- `upgradeToProVersion`

The `productPurchaseSucceeded:` method displays a `UIAlertView` to inform the user that the transaction has succeeded. The `mTableView` is reloaded to change the **Buy** button to a checkmark, and you can perform a test to see if the `productIdentifer` of the purchased product is a product you have defined to unlock certain features, such as an Upgrade to Pro version. In this example, the `productIdentifier net.yourdeveloper.MyStore.upgrade` indicated an Upgrade to Pro version product, which will remove advertisements.

Implement the `upgradeToProVersion` method, which simply removes the `surrogateAdvertisementView` from the `superView` and repositions the `UITableView` to a full screen.

The notification methods are shown below:

```
- (void)productPurchaseSucceeded:(NSNotification *)notification
{
      [self.mTableView reloadData];
      SKPaymentTransaction     *     transaction     =     (SKPaymentTransaction
      *)[notification.userInfo
                                    objectForKey:@"transaction"];
      UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Thanks!"
                                    message:@"for your purchase."
                              delegate:nil
                        cancelButtonTitle:nil
                        otherButtonTitles:@"OK", nil];
      [alert show];
      //Test if it's our upgrade product
      if ([ transaction.payment.productIdentifier
            isEqualToString:@"net.yourdeveloper.MyStore.upgrade"])
      {
      [self upgradeToProVersion];
      }
}
- (void)productPurchaseFailed:(NSNotification *)notification
{
SKPaymentTransaction      *      transaction      =      (SKPaymentTransaction
*)[notification.userInfo
                              objectForKey:@"transaction"];
if (transaction.error.code != SKErrorPaymentCancelled) {
      UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error!"

                              message:transaction.error.localizedDescription
                              delegate:nil
                              cancelButtonTitle:nil
                              otherButtonTitles:@"OK", nil];
      [alert show];
   }
```

```
}
- (void)deliverProduct:(NSNotification *)notification
{
      SKPaymentTransaction    *    transaction    =    (SKPaymentTransaction
      *)[notification.userInfo objectForKey:@"transaction"];
      //Test if it's our upgrade product
      if ([ transaction.payment.productIdentifier
            isEqualToString:@"net.yourdeveloper.MyStore.upgrade"])
      {
      [self upgradeToProVersion];
      }


            else
            {
                  //Deliver the product e.g. opening a level in a game or
                  download a PDF
            }
      }
      -(void)upgradeToProVersion
      {

      //remove the surrogateView from the superview
      [self.surrogateAdvertisementView removeFromSuperview];
      self.mTableView.frame = [[UIScreen mainScreen] bounds];
      [self.mTableView reloadData];


      }
```

You have now implemented the complete cycle of process steps required to display In-App Purchase products from an external source, display them in a user interface with a **Buy** button, perform the payment transaction, verify the payment receipt, and inform the user with the results.

Exam Check

In your certification examination, you will be required to develop In-App Purchases in your application.

Knowledge Check 2

- Q. No. 800
- Q. No. 821
- Q. No. 824

Quick Tip

The `YDInAppPurchaseManager` class should definitely be part of the Private Library project you developed earlier, if you aim to use the In-App Purchase functionality in your applications.

<H1> [Monetizing with Advertisements]

Source: [Professional IOS programming][Chapter 16][473]

Hundreds of advertising platforms are available for iOS applications. This section describes the implementation logic of the following two advertising platforms:

- **iAd**: This is the advertising platform from Apple.
- **AdMob**: This is the advertising platform from Google. It is one of the largest platforms available and has a dominant market position.

Big Picture

Just creating an application and dropping it to the App Store is not enough from the point of marketing strategy. Updating of the application is required if you want your audience to remain engaged with your application and earn money from it.

<H2> [Introducing the iAd Framework]

Source: [Professional IOS programming][Chapter 16][473-474]

The iAd framework does all the necessary work to download advertisements from the iAd network, which you can present on the user interface of your application. To use the iAd framework and display advertisements from the iAd network, you must agree to the iAd network license agreement, which is available in the member center of the developer portal. iTunes Connect is where you configure whether your application supports iAds.

Two types of advertisements are available:

- **Banner Views**: These advertisements use a portion of the screen to display a banner Ad.
- **Full-Screen Advertisements**: These advertisements are available only on the iPad and display a page of content served by the iAd network.

When a user taps the presented advertisement, it will typically cover the entire screen while the application continues running. As a result, the user cannot see the user interface of your application and you might want to postpone certain functionalities until the advertisement is no longer visible. You also have the option to cancel the displayed advertisement when your application requires the user's attention.

<H3> [Implementing iAd Advertising]

Source: [Professional IOS programming][Chapter 16][474-476]

Start Xcode and create a new project using the **Single View Application** template. Name your project as **MyiAdDemo** using the options shown in **Figure 7**.

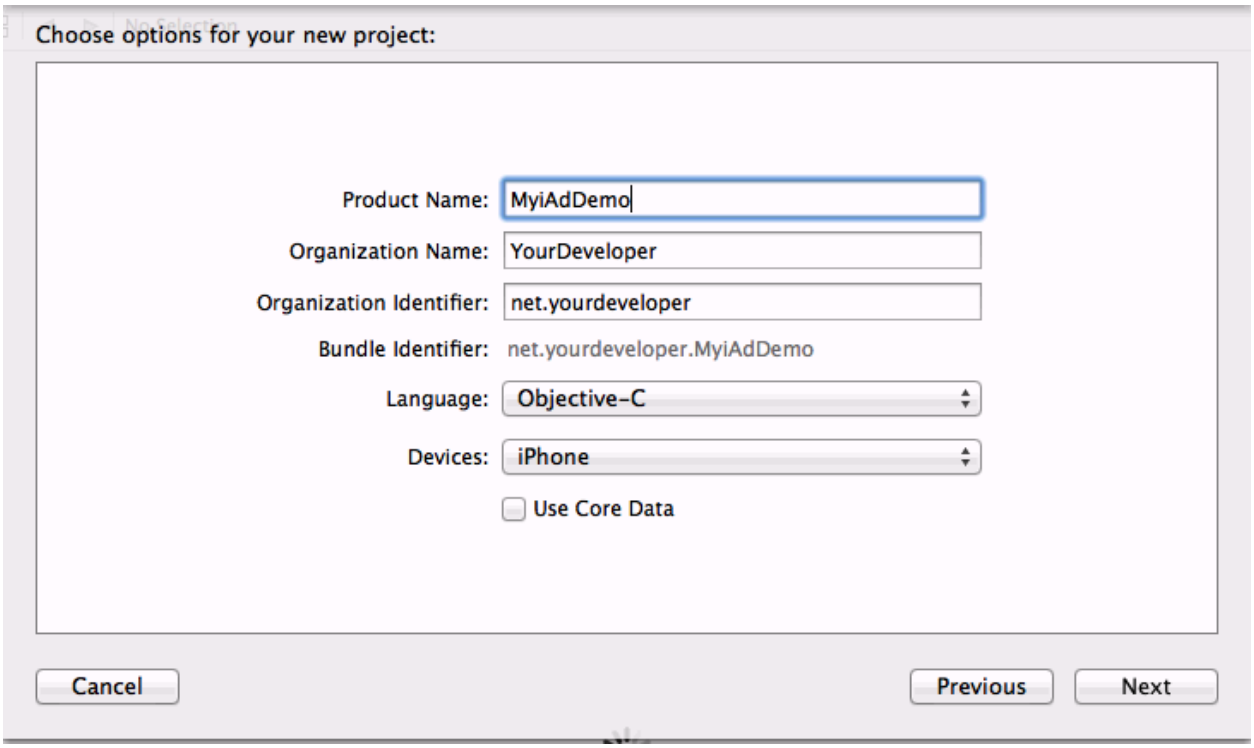


Figure 7: Create a New Project, MyiAdDemo, in Xcode

Add the iAd framework to your project, open the YDViewcontroller.m file, and import the iAd header file. Subscribe to the ADBannerViewDelegate protocol and create a private variable named adBannerview of type ADBannerView and a BOOL named bannerIsVisible. In the viewDidLoad method, create and initialize the adBannerView with a zero frame and call the setAutoresizingMask: of the adBannerView. Set the delegate property of the adBannerView to self and add the adBannerview as a subview to the current view.

If you launch your application now, an advertisement will display if the network can serve one. Since a network will never have a 100 percent fill rate (the fill rate is the ratio of advertisements served in relation to the number of requests made), a white frame might display instead of an actual advertisement.

Because you have subscribed to the ADBannerViewDelegate protocol and set the delegate to self, you need to implement the following methods:

- **The bannerViewDidLoadAd: delegate method**: This method is invoked once an advertisement is loaded
- **The bannerView:didFailToReceiveAdWithError: delegate method**: This method is invoked if the network cannot serve an advertisement.

Finally, you need to implement the willRotateToInterfaceOrientation:duration: method so the advertisement will display correctly when the user rotates the device.

The complete implementation of the YDViewController.m file is shown below:

```
#import "YDViewController.h"
#import <iAd/iAd.h>
@interface YDViewController ()<ADBannerViewDelegate>
{
      ADBannerView* adBannerView;
      BOOL bannerIsVisible;
}
@end

@implementation YDViewController
```

```
- (void)viewDidLoad
{
      [super viewDidLoad];
      // Do any additional setup after loading the view, typically from a
      nib.
      adBannerView = [[ADBannerView alloc] initWithFrame:CGRectMake(0,20,
      self.view.frame.size.width, 50)];
      [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
      adBannerView.delegate=self;
      [self.view addSubview:adBannerView];
}

- (void)bannerViewDidLoadAd:(ADBannerView *)banner
{
      bannerIsVisible=YES;
}
-                    (void)bannerView:(ADBannerView                    *)banner
didFailToReceiveAdWithError:(NSError *)error
{
bannerIsVisible=NO;
}

- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)
      toInterfaceOrientation duration:(NSTimeInterval)duration
{
      if (UIInterfaceOrientationIsLandscape(toInterfaceOrientation))
      [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
      else
      [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
}

- (void)didReceiveMemoryWarning
{
      [super didReceiveMemoryWarning];
      // Dispose of any resources that can be recreated.
}

@end
```

When you launch your application, you should get iAd banner on the opened app.

Quick Tip

You can find the iAd framework reference at *http://developer.apple.com/library/ios/#documentation/UserExperience/Reference/iAd_ReferenceCollection/_index.html*.

The iAd programming guide is available at *http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/iAd_Guide*.

<H2> [Implementing the AdMob Network]

Source: [Professional IOS programming][Chapter 16][476-477]

AdMob is the mobile advertising network from Google available on *www.admob.com*. You can create an account for free to become a publisher and you can start immediately.

Once you have created an account, you can create your application by selecting **Sites and Apps** from the drop-down navigation menu and clicking **Add Site/App**. Select the app type, which will be an iPhone App or an iPad App, and complete the details screen. When you click the **Save** button, the screen displays a button to download the SDK. Download the SDK and click the **Go to Sites/Apps** button, which presents you with the list of Sites/Apps available under your account.

Navigate to the application you have just created, and when you hover over that line, you will see two buttons: a **Reporting** button and a **Manage Settings** button. Click the **Manage Settings** button and you will see a Publisher ID under the application name. You need this Publisher ID to implement the AdMob SDK.

<H3>[Displaying AdMob Advertisements]

Source: [Professional IOS programming][Chapter 16][477-479]

To display the AdMob advertisements, open the `MyStore` project you created earlier and implement the display of AdMob advertisements as a replacement for the `surrogateAdvertisementView` method.

After you have downloaded the AdMob SDK and opened the `MyStore` project in Xcode, drag and drop the following files into your project:

- `libGoogleAdMobAd.a`
- `GADAdMobExtras.h`
- `GADAdNetworkExtras.h`
- `GADBannerViewDelegate.h`
- `GADInterstitial.h`
- `GADInterstitialDelegate.h`
- `GADRequest.h`
- `GADRequestError.h`
- `GADAdSize.h`
- `GADBannerView.h`

You need to add the following frameworks to your project to work with AdMob:

- `AdSupport`
- `SystemConfiguration`
- `MessageUI`
- `AudioToolbox`
- `CFNetwork`
- `StoreKit`

Finally, add `-ObjC` under Other Linker Flags in the **Build Settings** tab of your application.

Open the `YDViewController.m` file and import the `GADBannerView` header file. In the private interface, declare a variable named `bannerView` of type `GADBannerView`.

Change the `viewDidLoad` method as shown below:

```
- (void)viewDidLoad
{
      [super viewDidLoad];
      /*
      self.surrogateAdvertisementView=[[UIView alloc] initWithFrame:
                              CGRectMake(0,0,320,50)];
      self.surrogateAdvertisementView.backgroundColor=[UIColor blueColor];
```

```
      [self.view addSubview:self.surrogateAdvertisementView];
      */
      [self initializeInAppPurchaseManager];
      [self displayAdvertisement];
}
```

Create a new method named `displayAdvertisement`, which is responsible for displaying the AdMob advertisement. It starts by checking whether the user purchased the Upgrade to Pro product, which would remove the advertising. It does this by calling the `productPurchased:` method of the `YDInAppPurchaseManager` by passing the product identifier of the Upgrade to the Pro product. If the upgrade has not been purchased, the `bannerView` is created and initialized and added as a subview of the current view. Finally, the `loadRequest` method of the `GADBannerView` class is called, which is responsible for the communication with the advertising network.

The complete `displayAdvertisement` method is shown below:

```
-(void)displayAdvertisement
{
      if (![[YDInAppPurchaseManager sharedInstance]
                  productPurchased:@"net.yourdeveloper.MyStore.upgrade"])
            {
            // Create the GADBannerview
            bannerView = [[GADBannerView alloc] initWithFrame:
                              CGRectMake(0,0,
                                          GAD_SIZE_320x50.width,
                                          GAD_SIZE_320x50.height)];
            // Specify your AdMob Publisher ID.
            bannerView.adUnitID = @"a14f0070e2b8015";
            // Let the runtime know which UIViewController to restore after
            // taking the user wherever the ad goes and add it to the view
            // hierarchy.
            bannerView.rootViewController = self;
            [self.view addSubview:bannerView];

            GADRequest *request = [GADRequest request];
            request.testing=YES; // or NO
            // Initiate a generic request to load it with an ad.
```

```
            [bannerView loadRequest:[GADRequest request]];
            }
            else
            {
            self.mTableView.frame = [[UIScreen mainScreen] bounds];

            }
}
```

The `upgradeToProVersion` method needs a small change. Instead of removing the surrogate `AdvertisementView` from the `superView`, remove the `bannerView` from the `superView` as shown below:

```
-(void)upgradeToProVersion
{
      //remove the surrogateView from the superview
            //[self.surrogateAdvertisementView removeFromSuperview];
      [bannerView removeFromSuperview];
      self.mTableView.frame = [[UIScreen mainScreen] bounds];
      [self.mTableView reloadData];

}
```

Knowledge Check 3

- Q. No. 835
- Q. No. 839
- Q. No. 845
- Q. No. 859
- Q. No. 958
- Q. No. 967

Technical Stuff

The `displayAdvertisement` class is responsible for displaying the AdMob advertisement.

Quick Tip

The `GADBannerView` class is responsible for the communication with the advertising network. `GADBannerView` is used to create a banner for your application i.e. `bannerView =[[GADBannerView alloc]initwithframe] CSRectMake(0,0,GAD_SIZE_320 x 50. Width,GAD_SIZE_320x50.height );`

Additional Knowhow

Ads come in different dimensions and shapes. If you decide to put an ad on a particular screen in an application, you can add it in your activity's XML layout. You can use the XML snippet to add a banner ad displayed on the top of the screen.

Exam Check

In your certification examination, you will be required to monetize the advertisements in your application.

Lab Connect

During the lab hour of this session, you will be able to create paid applications with a commercial perspective.

Cheat Sheet

- Advertising platforms like Apple's iAd or Google's AdMob are used to generate revenue by displaying advertisements in your application.
- You can monetize your apps through the following ways:
    - Paid application
    - Advertising
    - In-App Purchases
    - Subscriptions
    - Lead generation
    - Affiliate sales
- The Store Kit framework communicates with the App Store on behalf of your application and provides localized information about the products you want to offer. Once a user makes a selection, your application uses Store Kit to collect payment from that user.

- You can sell the following kinds of products in the Store Kit:
    - Subscriptions
    - Services
    - Functionality
    - Content
- Before you can sell a product using In-App Purchases, you need to register it with the App Store through iTunes Connect.
- You can select from the following product types:
    - Consumable
    - Non-consumable
    - Auto-renewable subscription
    - Free subscription
    - Non-renewing subscription
- You can up-sell your app by locking pro-version functionality using In-App Purchases.
- The `SurrogateAdvertisementView` is used for flashing advertisements.
- If you want to buy or upload the application on the App Store, you must have an Apple account.
- You can use the following advertising platforms for iOS applications:
    - iAd: This is the advertising platform from Apple.
    - AdMob: This is the advertising platform from Google.