Payments

The Problem

The boss wants us to build a (digital) shopping cart component for a new command line app. To allow people to shop with the widest variety of payment methods, the payment module should allow multiple payment methods.

To start with, the payment module will just take *Paypal*, *Stripe* and *Amazon Payments*.

Later, the boss would like to add *ApplePay*, and *Google Wallet*.

In fact he would like the payment module to easily accommodate *any* payment method, even those not yet invented!

Learning Outcomes

- Can use and understand delegation
- Can understand protocols
- Can understand polymorphism

The Goal

- To create a command line app called *Payments* that models an on-line payment module
- The app will generate a random dollar value that simulates an online purchase and will display this to the user
- The user will be asked to select from 3 payment methods: *Paypal*, *Stripe* or *Amazon*
- Once they select a payment method the app processes the amount using the selected method
- We will use delegation to accomplish this

Setup Output and Input in `main.m`

Create a command line app called *Payments*.

When the app starts create a random dollar value between 100 and 1000. Use `arc4random_uniform()`.

Log a message to the user using the random integer that reads something like this:

`Thank you for shopping at Acme.com` `Your total today is $xxx` `Please select your payment method:` `1: Paypal, 2: Stripe, 3: Amazon`

Setup `fgets` to grab the user's input.

Convert the input to a primitive integer value. Log the input to the console for now. Run to test your work.

Creating the `PaymentGateway` Class

Let's start by creating the `PaymentGateway` class. Make it a subclass of `NSObject`.

Let's go ahead and give it just 1 method. Name this instance method something like `processPaymentAmount:`. It should take 1 `NSInteger` parameter and return void.

`main.m` is going to be the caller. So, let's import `PaymentGateway` into `main.m`. Instantiate it once you have parsed the user input.

Go ahead and call the `PaymentGateway`'s instance method `processPaymentAmount:`. Pass in the randomly generated dollar value.

`PaymentGateway` is the *Delegator*

The `PaymentGateway` class is going to be quite generic. This means it will have no specific knowledge of *concrete* payment classes. This makes it more *reusable*.

`PaymentGateway` cannot actually process payments itself. To do so it will need to reach out to one of the concrete payment classes, like `StripePaymentService`, `PaypalPaymentService`, or `AmazonPaymentService`. We will use delegation to do this.

`PaymentGateway` will be the *delegator*. The class that actually processes the payment will be the *delegate*. The delegate is going to be an instance of a concrete payment class.

Create 3 Concrete Payment Classes

So let's go ahead and create the 3 concrete payment classes. Let's call them something like `PaypalPaymentService`, `StripePaymentService` and `AmazonPaymentService`. They are all subclasses of `NSObject`.

Adding the `PaymentGateway` Protocol

`PaymentGateway` is delegating the payment task to one of these concrete payment classes. But it doesn't want to be tightly coupled to these.

To do this we will define a protocol in the header file of `PaymentGateway`. Let's call this protocol `PaymentDelegate`. Let's add a single method signature called `processPaymentAmount:`. It should return void and take an `NSInteger` parameter for the amount to process.

Adding the `PaymentGateway` `delegate`Property

We will also create a *weak* property of type *id*. Call the property *paymentDelegate*.

This property is of type *id* because we would like the responder to this method call to be anonymous. That is, the responder can be *any* class that implements the `PaymentDelegate` protocol.

If the responder was not anonymous then `PaymentGateway` might need to know which concrete payment class it was about to use. It might even need to call a *different* method on each concrete payment class to process payments!

> **Note:** Think about how much more complex being dependent on concrete payment types would make `PaymentGateway`. For one thing

it would make it messy to add new concrete payment types. By creating a loose coupling between `PaymentGateway` and its concrete payment classes we can add new concrete payment methods without having to change the `PaymentGateway` class!

Programmers call this the **open/closed principle**. This principle states that classes "should be open for extension, but closed for modification".

Conforming to the `PaymentDelegate`

Go ahead and make sure each of the concrete payment classes conforms to the `PaymentDelegate`protocol.

In the real world, each concrete payment class will most likely have a unique implementation even though every one of these 3 classes is called using the same protocol method. This is a good illustration of *polymorphism*.

To simulate a unique implementation of the protocol method, let's go ahead and log something different for each concrete payment class.

Hooking Everything up

In `main.m` make sure you've instantiated `PaymentGateway`.

Next, instantiate a concrete payment class according to the menu item the user selects. To do this convert the user's input to a primitive integer value and write a `switch` statement. Inside the body of each case of the `switch` statement you will want to instantiate the class corresponding to the case.

Assign this class instance to the `paymentDelegate` property of the `PaymentGateway` instance.

Next call `processPaymentAmount:` on the `PaymentGateway` instance and pass the randomly generated dollar value in the parameter.

Notice that `PaymentGateway` can't handle the actual processing itself. It just routes the processing request to the delegate. The delegate, in contrast, *can* process a payment.

So inside `processPaymentAmount:` call `self.paymentDelegate` and call the method defined in the protocol. Pass the dollar amount into the parameter of the method defined in the protocol. This calls the implementation of the protocol method on the concrete payment class you assigned to the protocol property.

Run your project. Confirm that everything is working correctly. Try selecting different payment methods.

Let's Add `canProcessPayment` to the Protocol

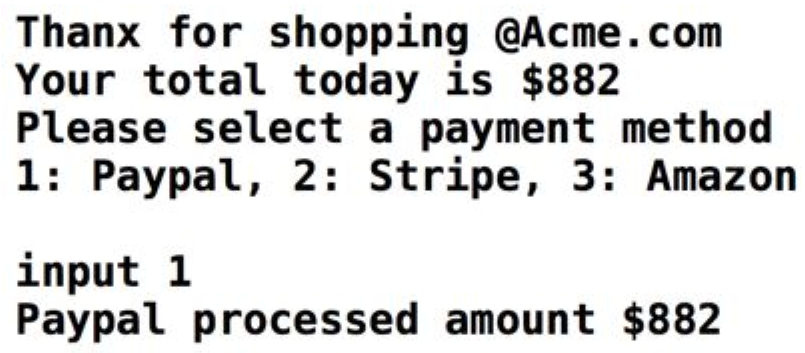
Since the concrete payment class might not be able to process the payment at this time, let's ask it to check if it can process the request before we actually send it the request.

To do this, add a method to the `PaymentDelegate`. This should return a BOOL. Let's name it `canProcessPayment`. Go ahead and add this method to the protocol.

Inside the method implementation on each of the concrete payment classes let's use `arc4random_uniform()` to generate either 0 or 1. If it generates 0 then return `NO`. If it generates 1 then return `YES`.

Now, add the additional logic inside the `PaymentGateway`'s `processPaymentAmount:` and check to see if the currently set concrete payment class can actually process the payment. If it can, then send the message. If it can't, then you can just output an apology to the user to inform them that their payment cannot be processed.

Your console output should end up looking something like this:



```
Thanx for shopping @Acme.com
Your total today is $882
Please select a payment method
1: Paypal, 2: Stripe, 3: Amazon

input 1
Paypal processed amount $882
```

Adding `ApplePaymentService`

The boss has finally got everything in place for ApplePay. So go ahead and add the `ApplePaymentService` class. Add the option to select `ApplePaymentService` to the initial menu of options. Notice that we do not need to make any change to `PaymentGateway` to add or remove any payment methods.

Reflection

At a very high level, delegation is simply getting some class to do some work for another class.

The delegate pattern in iOS uses protocols to do this. Protocols are a mechanism that limits the coupling between the delegator and delegate.

One of the reasons to limit coupling via a protocol is that it limits the disruption changes, which are inevitable in software development. In other words, if my `PaymentGateway` only depends on a method in the protocol, then we can easily remove or add new `PaymentService` types without impacting `PaymentGateway`. We could also change anything about the concrete payment classes without impacting `PaymentGateway` so long as the protocol method is not changed. If we were dependent on concrete `PaymentService` types changes to the concrete payment classes could require serious refactoring. In a simple example like this, it's not a big deal. But as a project gets bigger such changes can slow down development and eventually make adding features nearly impossible.

Programming to a protocol rather than an implementation simplifies our code. Some programmers call this principle the dependency inversion principle.

Most new developers coming to iOS struggle with the delegate pattern. But this pattern is absolutely everywhere in Apple's frameworks. So you will not be able to escape it.

Why `paymentDelegate` can be *strong* in our app?

A lot of the time delegation in iOS is used as a way to do "callbacks" . (We will get to this next week). But delegation is not limited to callbacks.

The `Payments` app is not currently using delegation to do a callback.

The reason *delegate* properties are *usually weak* in iOS is because they are usually used in callbacks. In a typical callback in iOS a controller *A* instantiates another controller *B*. *A* conforms to *B*'s protocol. *B* calls a protocol method on *A* using its *delegate* property. If A has a strong property to B when it instantiates it, and B has a strong property back to A via a *strong* delegate property then we would have a retain cycle *retain cycle*.

The *delegator* is `PaymentGateway`. The *delegate* is the concrete payment instance. Since the concrete payment doesn't call back to the `PaymentGateway`, the delegate doesn't need a reference to the `PaymentGateway`. So, there's no danger of a retain cycle. But `PaymentGateway` should have a strong reference to the concrete payment instance it is going to call.