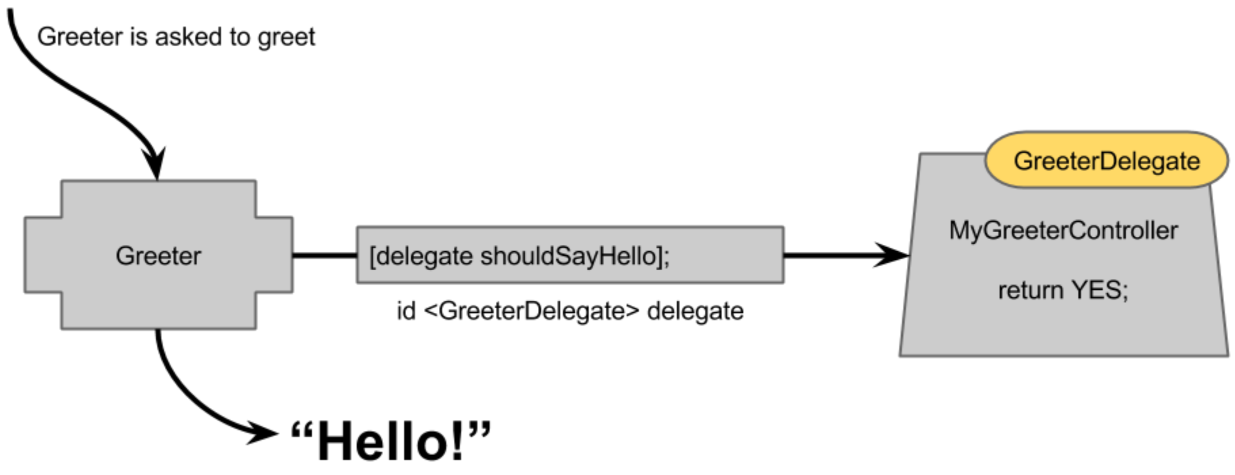
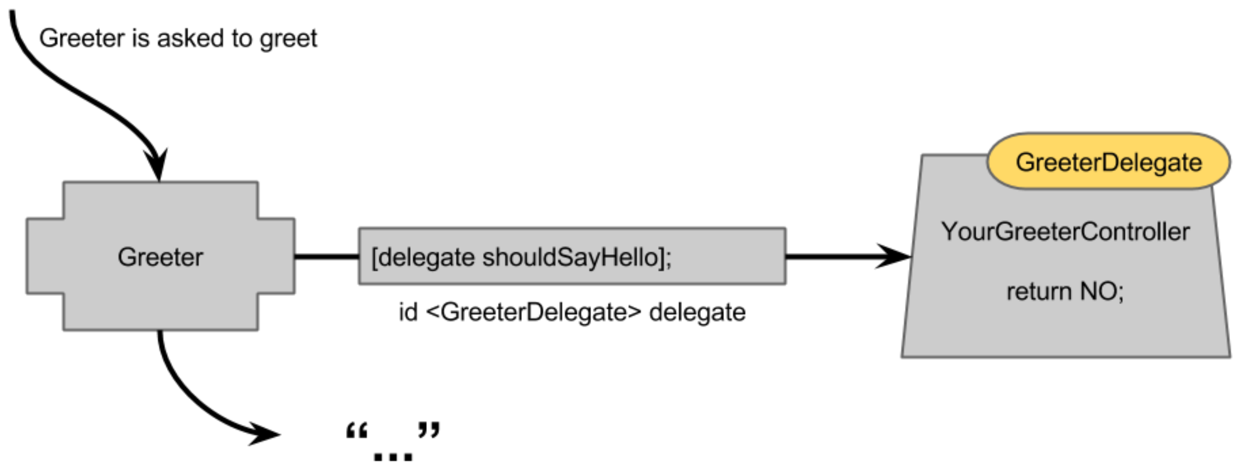
FoodTruck - Delegate pt. 1

Introduction

Delegation is a "design pattern", a way of arranging objects in an Object Oriented language. The goal of delegation is to build objects that a) can be *losely* connected to another object, and b) that other object can change the behavior of the first object.


Greeter is asked to greet
Greeter
[delegate shouldSayHello];
id <GreeterDelegate> delegate
GreeterDelegate
MyGreeterController
return YES;
"Hello!"

Shown above, the Greeter asks its delegate if it should say hello. MyGreeterController always returns YES when asked if a greeting should happen, so the greeter says "hello".


Greeter is asked to greet
Greeter
[delegate shouldSayHello];
id <GreeterDelegate> delegate
GreeterDelegate
YourGreeterController
return NO;
"..."

Shown above, a different delegate is connected to our Greeter, this one always returns NO when asked, so the greeter does *not* say hello.

In both examples the *delegator* is on the left and the *delegate* is on the right.

The big picture

Below you'll see the list of steps required to implement a delegate from scratch. **We will *not* be doing it from scratch in this assignment.**

Steps A, B, and C have been done already, for this assignment we'll be doing only steps D, E, and F.

A. Create a Protocol that defines what messages the delegate can receive.

```
@protocol GreeterDelegate
-(BOOL)shouldSayHello;
@end
```

B. Add a delegate property to your delegator

```
@property (nonatomic, weak) id<GreeterDelegate> delegate;
```

C. From your delegator, send messages to your delegate.

```
[self.delegate shouldSayHello];
```

D. Adopt the protocol in the definition of a class you've created.

```
@interface FriendlyGreetingDecider : NSObject <GreeterDelegate>;
```

E. Implement how your delegate responds to the methods in the protocol.

```
-(BOOL)shouldSayHello {
  return YES;
}
```

F. Connect your delegator and delegate together.

```
greeter.delegate = greetingDecider;
```

In this example, we've created a system where a `Greeter` class can be loosely connected to anything that implements the `GreeterDelegate` protocol. Once connected, the `Greeter` asks its delegate, in this case our

`FriendlyGreetingDecider`, if it should say hello. If the delegate responds `YES` it will greet, if it responds `NO` it doesn't greet.

Part 1 - Becoming a delegate

Download the starter proj, and open up the project.

1. Create a new class called `Cook`, this will be our delegate.
2. Make `Cook` conform to the `FoodTruckDelegate` protocol. This is step D in our delegation process.
3. Make `Cook` implement the required methods defined in `FoodTruckDelegate`. For now just make it a simple implementation that always returns the same thing. This is step E.
4. In your `main.m` file, create an instance of `Cook`.
5. Connect your instance of `Cook` to the instances of `FoodTruck` via their `delegate` property. This is step F.

Run the app and test the behaviour of the object.

Why did we do it this way

We could have wired `Cook` and `FoodTruck` together directly, but we went through all these extra steps because we wanted their connection to be *loosely coupled*. This means the job `Cook` does for `FoodTruck` can be filled by anything that follows the set of "rules" in `FoodTruckDelegate`.

This becomes especially useful when we don't control `FoodTruck`. For example because it was written by someone else as part of a framework. We'll see delegates pop up all over UIKit as a way to make things like text fields or table views behave differently in different parts of our app.

Part 2 - Another delegate

Let's make another delegate class and connect it to our `FoodTruck`. This time make it cause the `FoodTruck` to charge different prices depending on what food type is being served. Its implementation of `-foodTruck:priceForFood:` is different, so when it is the `FoodTruck`'s delegate the truck behaves differently.

Notice that we will not have to modify `FoodTruck` at *all* to change what it does. Mission accomplished!