# Basic Exercises Part 9.3 Protocols

## Protocols in Objective-C

* In iOS development, a protocol is a set of methods and properties that encapsulates a unit of functionality. The protocol doesn’t actually contain any of the implementation for these things; it merely defines the required elements. Any class that declares itself to conform to this protocol must implement the methods and properties dictated in the protocol.

### **1.1 Declaring a protocol**

In Objective-C, protocols are declared with the @protocol keyword.

# Swift

protocol SampleProtocol {

func someFunction()

}

# Objective-C

@protocol SampleProtocol

- (void) someMethod;

@end

### **1.2 Conforming to a protocol**

You add the protocol name in angle brackets beside the class interface declaration. “MyTestClass” is declaring that its conforms to SampleProtocol. MyTestClass will also have to provide an implementation for “someMethod” in the implementation file because it is a required protocol method.

# Objective-C

@interface MyTestClass : NSObject

@end

In Swift, the protocol name is appended beside the superclass name, separated with a comma. If there isn’t a superclass, the you just write a colon, followed by the protocol name.

# Swift

class MyTestClass: SampleProtocol {

// conforming to SampleProtocol

func someFunction{

// … code

}

}

class SecondTestClass: ClassA, SampleProtocol {

// a subclass conforming to SampleProtocol

func someFunction{

// … code

}

}

### **1.3 Delegation**

Delegation works hand in hand with protocols because it allows a class to specify a delegate property which conforms to some protocol. Then a second class which actually conforms to that protocol can be assigned to that property. Now the first class can communicate with the second class through the delegate property using the methods and properties as defined by the protocol.

### **1.4 Declaring a delegate property**

In Objective-C, declaring a delegate property involved using the “id” keyword as shown below.

# Objective-C

@interface MyTestClass : NSObject

@property (nonatomic, weak) id delegate;

@end

In Swift, declaring a delegate property is just like declaring any other property and you specify the protocol name as the type of the property. You may notice the question mark syntax which indicates that it’s a property with an optional value (there may or may not be an object assigned to it).

# Swift

class ClassA {

var delegate: SampleProtocol?

}

### **1.5 Calling a delegate method**

In Objective-C, often time, you would see an IF statement to check if there was an object assigned to the delegate property before calling the delegate method.

if (self.delegate) {

[self.delegate someMethod];

}

In Swift, you can take advantage of the question mark syntax. If the delegate property is empty, nothing after the question mark will be executed.

delegate?.someFunction()

### **1.6 Exercises**

Make this example works in your current project. Assume you have this file as lib.

TestLib.h

#import <Foundation/Foundation.h>

@protocol TestLibDelegate<NSObject>

-(void)updateCount:(int)count;

@end

@interface TestLib : NSObject

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

-(void)startUpdatingCount;

@end

TestLib.m

#import "TestLib.h"

@implementation TestLib

-(void)startUpdatingCount{

int count = 0; //Create count

if ([self.delegate respondsToSelector:@selector(updateCount:)]) {

[self.delegate updateCount:count];

}

}

@end

Then in the class you want to use

#import "ViewController.h"

#import "TestLib.h"

@interface ViewController ()<TestLibDelegate>

@property (strong,nonatomic)TestLib \* lib;

@end

@implementation ViewController

-(void)viewDidLoad{

self.lib = [[TestLib alloc] init];

self.lib.delegate = self;

[self.lib startUpdatingCount];

}

-(void)updateCount:(int)count{

NSLog(@"%d",count);

}

@end