Maths - Part 2 Goals

- The app should report the total play time. It should also report the average time it takes the player to answer a question. It should look something like this:

```
total time: 60s, average time: 10s
```

* Add other basic math operations like subtraction, multiplication, division

Part 2 Learning Outcomes

- Can override a property getter
- Can calculate the difference between two `NSDate`s and return an `NSTimeInterval`
- Can understand and implement a model manager class that holds an array of model instances. This class exposes methods and properties to manipulate the array
- Can call methods on `self`
- Can understand and implement subclassing
- Can understand and implement a class that makes other class instances (*factory pattern*)
- Can understand `super` vs `self`
- Can understand and use polymorphism

Adding Start and End Times to `AdditionQuestion`

We want to output the total time and average time.

To do so we need to decide which object should be responsible for tracking the time. We could make a separate class for tracking the time. But to keep it simple let's have `AdditionQuestion` handle this responsibility.

So, let's go ahead and give the `AdditionQuestion` 2 properties. They should have names like `startTime` and `endTime`. Make them of type `NSDate`.

Next we have to think about where in our code we should set the `startTime`. A reasonable option is in `AdditionQuestion`'s `init` override. The moment of object creation corresponds to the `startTime`. So go ahead and set `_startTime` to the current date. Use the convenience initializer `[NSDate date]`.

Next we have to think about where to set the `endTime` property. We could set it from `main.m` at the moment the user answers the question. This is a reasonable option. It would allow us to pass in a dummy date value to test the method if we wanted to. This is called "Dependency Injection".

Instead of passing in the `endTime` let's override the `answer` getter. When `main.m` calls `AdditionQuestion` for the `answer`, we can set the `endTime` then. Remember Objective-C properties are just getter and setter methods. These are automatically generated by the compiler. So, let's go ahead and override the `answer` getter and set the `endTime` property like this:

```
// overriding getter
- (NSInteger)answer {
```

```
    _endTime = [NSDate date];
    return _answer;
}
```

Note when we intercept calls to get the answer we must still return `_answer`.

Calculating the Time Difference

OK we also need to know the time difference between `startTime` and `endTime` in seconds. We will need to add a method to `AdditionQuestion`. Let's call this method `answerTime`. This method should *return* a calculated value based on the `startTime` and `endTime`. The return type is an `NSTimeInterval`.

To do the calculation use the `NSDate` method `timeIntervalSinceDate:`. Look up how to use it here. You will want to round this value to make it more human readable.

> **Note:** Date and calendrical programming are tricky. To investigate this check out this tutorial.

Creating a Question Manager

We made `startTime` and `endTime` the responsibility of `AdditionQuestion`. But these properties don't yet give us what we are looking for: *total time*, and *average time*. We need to compute these values from all the `AdditionQuestion`

instances we have initialized. To do so we will need to have a collection of all the instances of `AdditionQuestion` as we create them.

We can't manage a collection of `AdditionQuestion`s inside `AdditionQuestion` without confounding its responsibility.

We could add an `NSMutableArray` to `main.m` and do everything in `main.m`. But again we are trying to avoid creating a bunch of spaghetti code.

What we need is another object to handle the collection of question instances. This object will also expose methods that need access to all question instances. Total time and the average answer time are just such methods.

> **Note**, this is actually a basic design pattern you will be using all the time. Chapter 28 of the book *Cocoa Design Patterns* calls this the `Manager Pattern`.

OK so let's go ahead and create a class called `QuestionManager`. Give it a property of type `NSMutableArray` called `questions`. You will need to initialize this property to an empty array. Do so by overriding the `QuestionManager`'s `init` method.

Next go ahead and instantiate `QuestionManager` outside `while` in `main.m`.

When you create an `AdditionQuestion` instance in `main.m` remember to add it to the `questions` array`.

Generating the Timing Output

`QuestionManager` is also going to generate a string like this: `total time: 60s, average time: 10s`

So go ahead and create a method called something like `timeOutput`. Make it return an `NSString*`. Generate this string and return it.

In `main.m` call this method. Log the returned string. Your app should now be able to produce an output like this:

```
2016-09-09 17:08:23.109 Maths[41543:740709] MATHS!


2016-09-09 17:08:23.110 Maths[41543:740709] 16 + 85 ?
101
2016-09-09 17:08:30.924 Maths[41543:740709] Right!
2016-09-09 17:08:30.924 Maths[41543:740709] score: 1 right, 0 wrong ---- 100%
2016-09-09 17:08:30.924 Maths[41543:740709] total time: 8s, average time: 8s
2016-09-09 17:08:30.925 Maths[41543:740709] 6 + 82 ?
68
2016-09-09 17:08:41.339 Maths[41543:740709] Wrong!
2016-09-09 17:08:41.339 Maths[41543:740709] score: 1 right, 1 wrong ---- 50%
2016-09-09 17:08:41.339 Maths[41543:740709] total time: 18s, average time: 9s
2016-09-09 17:08:41.339 Maths[41543:740709] 99 + 75 ?
```

Creating a Class Hierarchy

We could have completely separate classes for each math operation. But there would be a lot of repeated code. Each class would have to repeat code to do all the following: 1) generate 2 random values, 2) calculate and set an `answer` property, 3) compute an `NSString*` and set the `question`property, 4) track the start and end time, 5) compute the duration it takes to answer the question.

We want to avoid repeating the same code in 4 different classes! Repeated code is a maintenance nightmare.

One way to reuse code is to use class *inheritance*. We can create a superclass and put all the shared code there.

> **Note:** If you are uncertain about inheritance in Objective-C please review this video:
>
> https://youtu.be/B23RUDw7pjg

Let's go ahead and create a superclass called `Question`. One simple way to do this is to just rename `AdditionQuestion` to `Question`. After all most of what is in `AdditionQuestion` will make up our superclass. We will need to refactor the old `AdditionQuestion` though. So go ahead and rename `AdditionQuestion` to `Question`.

My `Question` class's header looks like this after renaming:

```
@interface Question : NSObject
@property (nonatomic, strong) NSString *question;
@property (nonatomic) NSInteger answer;
@property (nonatomic, strong) NSDate *startTime;
@property (nonatomic, strong) NSDate *endTime;
- (NSTimeInterval)timeToAnswer;
@end
```

Next create 4 classes that subclass `Question`. Name them something like `AdditionQuestion`, `SubtractionQuestion`, etc.

So, we need to think about what behaviour and data the superclass should share. Actually, it might be easier to think about what behaviour and/or data is *unique* to each subclass. The only thing unique to each subclass is that they generate a string with an operator unique to their class. They will also calculate their answer using this unique math operator.

So, let do this: We can add 2 new properties to `Question` called `rightValue` and `leftValue`. These will be set in `Question`'s `init` override to random values. Each subclass will be able to access these random values. They can use these to build up their unique question string and compute their answer.

What we also need is a method for setting the `question` and `answer` properties. We should declare this method in the superclass. But we should create a unique implementation to the method in each subclass.

Let's go ahead and add a method to `Question` called `generateQuestion` that returns `void`.`Question`'s implementation of `generateQuestion` will do nothing. So leave it blank.

Each `Question` subclass will override `generateQuestion`. In their implementation they will set the superclass's `answer` and `question` property. To do so they will use super's `leftValue` and `rightValue` random values.

For example, the `SubtractionQuestion`'s `generateQuestion` method will set the superclass's `question` property to `23 - 20 ?`. (23 and 20 are just some random values generated by the superclass).

To call `generateQuestion` each subclass can override `init` and call the method there. Your subclass implementations should look something like this:

```
@implementation SubtractionQuestion
- (instancetype)init {
    if (self = [super init]) {
        [self generateQuestion];
    }
    return self;
}

- (void)generateQuestion {
    // set super.answer here
    // set super.question here
}
@end
```

The `Quesition.h` should now look something like this:

```
@interface Question : NSObject
@property (nonatomic, copy) NSString *question;
@property (nonatomic) NSInteger answer;
@property (nonatomic, strong) NSDate *startTime;
@property (nonatomic, strong) NSDate *endTime;
@property (nonatomic) NSInteger rightValue;
@property (nonatomic) NSInteger leftValue;
```

```
- (NSTimeInterval)timeToAnswer;
- (void)generateQuestion;
@end
```

Creating Random Questions (Factory Pattern)

When we only had addition questions to worry about we didn't need to think about generating a random question *type*. Now we do. We need to randomly generate one of our subclasses of `Question`.

We should create a class to handle this! The responsibility of this class is simple. It is going to randomly generate and return an instance of 1 of the 4 subclasses of `Question`. This is a classical design pattern called the "factory pattern".

To do this create a class called `QuestionFactory`. We only actually need to instantiate this factory object once. So do it outside `while` in `main.m`. It should just have a single method called something like `generateRandomQuestion`.

Here's something a bit surprising though. The return type of this method should not be one of the 4 *concrete subclasses* of `Question`! Instead, the return type should be *upcasted* to the *superclass* `Question`.

Under the hood, the returned object will be a *concrete subclass* of `Question`. But making the caller aware of the actual subclass is unnecessary. In fact knowledge of the concrete subclass may complicate our code.

We only need a particular concrete subclass to generate the `question` and `answer` properties. `main.m` (the caller) will access everything it needs from properties and methods on the superclass.

This concept of upcasting to a supertype lies behind many Object Oriented Patterns. It also lies at the root of *polymorphism*.

A simple way to randomly generate a subclass of `Question` is to create an `NSArray` of `NSStrings`. Simply make this an `NSArray` literal with string literals for each class name. Like this:

```
NSArray *questionSubclassNames = @[@"AdditionQuestion",
@"SubtractionQuestion",...];
```

> **Note:** This array of strings will never change. So you might want to initialize it by overriding `QuestionFactory`'s `init` method.

Pick a random string by generating a random integer value to access the string array by index. Then take this class name string and instantiate the class. You can do this using the method `[[NSClassFromString(className) alloc]init];`.

Once you're done, your console output should look something like this:

```
2016-09-10 19:40:43.935 Maths[17574:340674] Right!
2016-09-10 19:40:43.935 Maths[17574:340674] score: 8 right, 2 wrong ---- 80%
2016-09-10 19:40:43.935 Maths[17574:340674] total time: 104s, average time: 10s
2016-09-10 19:40:43.935 Maths[17574:340674] 98 + 63 ?
4
2016-09-10 19:40:48.988 Maths[17574:340674] Wrong!
2016-09-10 19:40:48.989 Maths[17574:340674] score: 8 right, 3 wrong ---- 73%
2016-09-10 19:40:48.989 Maths[17574:340674] total time: 109s, average time: 9s
2016-09-10 19:40:48.989 Maths[17574:340674] 70 - 14 ?
56
2016-09-10 19:41:02.305 Maths[17574:340674] Right!
2016-09-10 19:41:02.305 Maths[17574:340674] score: 9 right, 3 wrong ---- 75%
2016-09-10 19:41:02.305 Maths[17574:340674] total time: 122s, average time: 10s
2016-09-10 19:41:02.305 Maths[17574:340674] 86 * 1 ?
```

The top of my `main.m` now looks like this:

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        BOOL gameOn = YES;
        NSLog(@"MATHS!\n\n\n");
        NSString *right = @"Right!\n";
        NSString *wrong = @"Wrong!\n";
        ScoreKeeper *scoreKeeper = [[ScoreKeeper alloc] init];
        InputHandler *inputHandler = [[InputHandler alloc] init];
        QuestionManager *questionManager = [[QuestionManager alloc] init];
        QuestionFactory *questionFactory = [[QuestionFactory alloc] init];

        while (gameOn) {
            // do stuff here
        }
    }
    return 0;
}
```