Maths - A Command Line Game App

Part 1 Learning Outcomes

- Can understand *class* vs *instance* method
- Can understand the difference between a method *signature*, the method *definition* and calling a *method*
- Can decide when to override `init` versus creating a custom initializer
- Can override `init`
- Can understand why we set the "backing store" of properties inside `init` using `_` (underscore) rather than calling `self`
- Can strip white space and new characters from string using `NSCharacterSet`
- Can call a class convenience method on `NSCharacterSet`
- Can generate random integers between a given range using `arc4random_uniform()`
- Can convert `NSString*` to a primitive `NSInteger`
- Can understand how to modularize functionality into separate classes
- Can refactor code to move functionality into a class
- Can decide when to use a *class* vs an *instance* method

Part 1 Goal

- To create a command line game called *Maths* that will generate a random addition question
- To prompt the user to input their answer
- To parse the user's inputted answer and convert it to a primitive `NSInteger`
- The app will log "Right!" for correct and "Wrong!" for incorrect answers
- The app will present the *next* question immediately after the app outputs the evaluation of the user's input (for now there's no exit option)
- To add the ability to exit the game
- To add a scoring function to the game

Instructions

Start by creating a command line app called *Maths*. Let's create a while loop in `main.m` to prevent the app from just exiting.

Converting a C String to an NSString

We're going to get the user input using `fgets`. We want to work with Objective-C rather than `C`. So we need to initialize an `NSString*` with the `C` string we get from `fgets`. To achieve this we can use the `NSString`

convenience initializer `+ stringWithCString:encoding:`. Please see the documentation.

Note: In yesterdays assignment Word Effects our Input Collection solved the same problem. It is an important concept to grasp that you can solve the same problem different ways.

Instance vs Class Methods

Notice this initializer starts with a `+` symbol. This means it is a *class* method, as opposed to an *instance* method. (If the distinction between *class* and *instance* methods is unclear, read this).

Understanding Apple's Documentation

In Apple's documentation `+ stringWithCString:encoding:` is defined in two distinct ways. The way I just mentioned, Apple calls the "method signature" or "selector". This is a terse way of referring to a method. It is the method with the parameters and return types omitted.

Apple also defines the method in a longer style. This is how the method looks from the perspective of the caller. It looks like this:

```
+ (instancetype)stringWithCString:(const char *)cString
                         encoding:(NSStringEncoding)enc
```

Notice that this longer definition tells us the parameter types. It also tells us the method's return type.

Apple doesn't usually tell us how to call the method. We have to figure this out. Since this is a *class* method we invoke it on the class `NSString` (not on an instance of `NSString`). We also need to pass it 2 parameters. The first is a `const char *` or C string (this is what we grab from `fgets`). The second parameter we need to pass it is an `NSStringEncoding` value which is an `enum`. So, here's how we call it to get an NSString instance which I'm calling "result".

```
NSString *result = [NSString stringWithCString:someCString
                                      encoding:NSUTF8StringEncoding];
```

Note: If you look at the documentation there is also an *instance* version of the same method. How would you call the instance version? Which version is preferrable and why?

Parsing User Input

When `fgets` grabs the input from the console it includes the new line character. We add this when we press “enter” after inputting text. There are many ways we could remove it. But we’re going to do it the right way. We will

use the NSString method called `stringByTrimmingCharactersInSet:` to remove it.

If we look at the documentation this method takes a parameter of type `NSCharacterSet`. It returns a new `NSString` instance .

`NSCharacterSet` has a bunch of *convenience* (class) initializers that handle the most common cases. In the documentation Apple lists these under the heading "Creating a Standard Character Set".

In our situation we should initialize an `NSCharacterSet` with the convenience initializer `+ whitespaceAndNewlineCharacterSet`. This way we will remove both any leading spaces the user may have entered and the new line.

It's a good idea, at this point, to just output the parsed `NSString*` to the console. This way we can test that everything is working.

**Tip:* *I recommend always working in small increments like this then test before moving to the next step. This limits problems from multiplying and overwhelming you.

Generating a Random Question

It's time to get to the heart of the app and write the code to generate a random addition question.

We could just dump all this logic into `main.m`. But we want to avoid creating a bunch of hard to understand spaghetti code! It is better to isolate this functionality into a separate class.

Let's give our class a clear, simple and descriptive name like `AdditionQuestion`. The responsibility of this class will be to generate a random math question. Also, this class will be responsible for handling the *answer* to the question. This makes sense given the class already has knowledge of the question and can compute the answer.

Think about how you would structure this class. If you feel ready, try to create the class on your own and then come back to the instructions. Otherwise keep following.

Overriding `AdditionQuestion`'s `init`Method

`AdditionQuestion` could have a method like `generateRandomQuestion`. The other alternative is to generate a random question right when we instantiate `AdditionQuestion`. Think about which is a better design then read below.

Requiring the caller to call a method *in addition* to instantiating the class is an extra step. By generating a random question as part of the initialization we can omit this unnecessary step.

The other thing to consider is whether we should override the default initializer `init`. Or should we create a custom initializer? The rule is to only create a custom initializer if we need to pass a parameter in during initialization. Do we need to do that? We *could* pass the 2 random values `AdditionQuestion`needs via a custom initializer. This is not a bad idea. But for simplicity's sake let's encapsulate random number generation inside the `AdditionQuestion`'s 'init' method. There's no need to pass anything in during initialization. So, let's go ahead and override `init`.

```
// AdditionQuesiton.m

- (instancetype)init {
    if (self = [super init]) {
    // do something here!
    }
    return self;
}
```

One goal of our override is to generate a random addition question as an `NSString*`. That way `main.m` can output this question string to the console when the user rolls.

We also need an `answer` property, which is the result of summing the 2 random numbers. In `main.m`we will compare the user input to the `answer`. We can then output whether the user got the answer right/wrong.

So, let's go ahead and create 2 properties. We will make the first one an `NSString` and call it something like `question`. The other property should be an `NSInteger` called `answer`. Put these in the `.h` file of `AdditionQuestion`.

Generating Random Numbers

Inside `init` let's generate 2 random numbers between 10 and 100 for the left and right side of the addition expression.

It is best to use `arc4random_uniform()` for generating random numbers.

Once we have 2 random numbers we can generate an NSString* expression something like `10 + 40 ?`. Then let's go ahead and assign it to the `question` property.

Tip: Inside `init` set the properties using `_` and not `self`, since `self` is still under construction. The `_` accesses the property's stored value directly and does not call the compiler generated setter and getter methods.

Recall that properties in Objective-C are just methods under the hood with stored values. If this is unclear to you read this excellent discussion of properties.

Finally set the `answer` property. Do this by summing the 2 random values together inside `init` and assigning them to `_answer`.

Connecting Everything

Inside the while loop of `main.m`, create an instance of `AdditionQuestion`. Grab the `question` string from the new instance and log it.

Grab the user input. Convert it from an `NSString*` to an `NSInteger` using the property `intValue`. Compare this value to the question instance's `answer`. Log out the message `Right!` or `Wrong!`depending on whether they got it right or wrong.

Your console should look something like this:

```
2016-09-07 22:21:10.027 Maths[97219:1701774] MATHS!


2016-09-07 22:21:10.028 Maths[97219:1701774] 98 + 13 ?
111
2016-09-07 22:21:29.048 Maths[97219:1701774] Right!
2016-09-07 22:21:29.048 Maths[97219:1701774] 23 + 62 ?
2
2016-09-07 22:21:34.457 Maths[97219:1701774] Wrong!
2016-09-07 22:21:34.457 Maths[97219:1701774] 100 + 16 ?
```

Adding Quit

Let's start by adding the quit functionality to the app. To do this we are going to check to see if the user's input matches the string "quit". If it does, we will break out of the while loop.

To do this you can set a BOOL variable outside the while loop to `YES`. Call it something descriptive like `gameOn`. If the user's input is "quit" then set this variable to `NO` and call `continue`.

We could also just call `break` which will force us to jump out of the while loop.

Note: In loops `break` exits a loop completely. `continue` jumps to the next iteration.

Moving Input Handling to a Separate Class

Next let's refactor our code to keep `main.m` clean. It's better to have a separate class handle the input and just return a parsed string to `main.m`. By modularizing our code we make it easier to understand, maintain and test.

Move input handling to a separate subclass of `NSObject` called something like `InputHandler`. We're going to need at least 1 method on this class that will use `fgets` and return the parsed `NSString*`.

Question: Should this be an *instance* method or a *class* method? Try creating the method in both ways. Is there a reason to prefer one over the other?

Finally, import this new class into `main.m`. Test to make sure everything is still working correctly.

Adding Scoring

Again, we're going to modularize the scoring functionality into a separate `NSObject` subclass. Let's call this class `ScoreKeeper`.

Think about the job of this class. It's going to have to track the number of right and wrong answers. It will also need to generate a string representation of the score for logging.

So, we will need 2 properties and 1 method. The properties will track the right and wrong counts. The method will output a string like this: `score: 3 right, 2 wrong ---- 60%`

You will need to import and instantiate `ScoreKeeper` in `main.m`. Run and test your work.