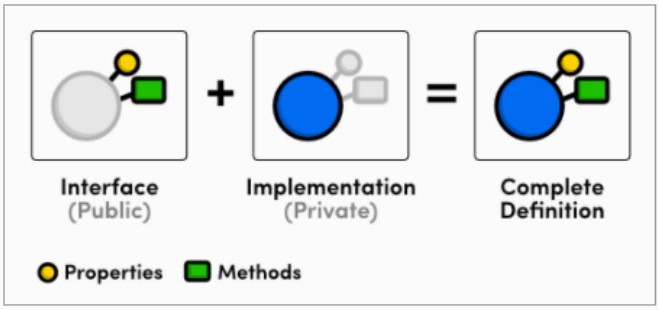
TEMAT: Objective-C.

WPROWADZENIE

1. Programując aplikacje mobilne na platformę iOS warto zapoznać się z dwoma językami bazowymi, zarówno Objective-C, jak i Swift. W ramach niniejszej pracowni specjalistycznej z Systemów Mobilnych będziemy omawiać jeden z nich: **Objective-C.**
2. **Zmienne –** standardowa składnia deklarowania zmiennych: <typ> <nazwa>. Przypisanie wartości za pomocą znaku =, np. `int liczba = 3;`
3. **Łańcuchy znakowe –** tworzone z użyciem @, np. `@"Przykładowy tekst"`.
4. **Stałe –** definiowane przy pomocy słowa kluczowego const. Przykład: `double const e = 2.71828;`
5. **Makra –** inna metoda definiowania stałych. Wykorzystywane jest słowo kluczowe #define. W tym wypadku tuż przed parsowaniem kodu wszystkie wystąpienia zdefiniowanej stałej są zamieniane na przypisaną jej wartość,
   np. `#define PI 3.14159`
6. **Struktury (structs) –** proste obiekty, które posiadają jedynie pola, nie mają zaś żadnych metod.
7. **Typ wyliczeniowy (enum) –** zbiór powiązanych stałych. Wykorzystywane w przypadku, gdy jakaś zmienna można przyjmować określoną i ograniczoną listę wartości.
8. **Wskaźniki –** bezpośrednie odwołanie do miejsca w pamięci (adresu).
9. **Funkcje -** muszą być deklarowane przed ich pierwszym użyciem. Możliwe jest rozdzielenie deklaracji od implementacji.
10. Implementacja **klas** w języku Objective-C wiąże się z koniecznością przestrzegania określonych reguł, zawierających wiele założeń spotykanych w innych językach programowania.



Poniżej opisane zostały najbardziej istotne zagadnienia związane z tworzeniem klas:

a. **Interfejs** – zawiera deklaracje publicznych właściwości (pól i metod) danej klasy. Jest to plik z rozszerzeniem `.h` o nazwie odpowiadającej nazwie klasy, której definicję posiada. Plik interfejsu powinien zawierać przede wszystkim dyrektywę `@interface`, nazwę klasy oraz poprzedzoną dwukropkiem nazwę klasy nadrzędnej (standardowo jest to NSObject). Oprócz tego kod umieszczony przed dyrektywą `@end` powinien zawierać definicje pól i metod. Ważne: *w przypadku konieczności wykorzystania danej klasy w innej klasie należy zaimportować wyłącznie jej interfejs*.

```
// Product.h
#import <Foundation/Foundation.h>

@interface Product : NSObject

@property (copy) NSString *name;
@property double price;

- (void)increasePrice:(double)percentage;

@end
```

b. **Pola** – deklarowanie pól w interfejsie odbywa się poprzez wykorzystanie słowa kluczowego `@property`, po którym należy podać standardową definicję zawierającą typ oraz nazwę pola. Dzięki dedykowanej dyrektywie `@property` pole otrzymuje niejawnie przypisane metody getter oraz setter, które (chociaż niewidoczne w pliku interfejsu) mogą być swobodnie używane z poziomu innych klas. Nie ma więc konieczności jawnego implementowania metod dostępowych. Domyślne metody getter i setter korzystają z następującej konwencji nazewniczej:

```
- (double)price {
    return _price;
}

- (void)setPrice:(double)newValue {
    _price = newValue;
}
```

W ten sposób pobranie pola klasy odbywa się poprzez podanie jego nazwy, zaś przypisanie nowej wartości wymaga poprzedzenia nazwy pola słowem kluczowym `set`. Istnieje jednak jeszcze inna możliwość dostępu do pola: z użyciem operatora kropki (*dot-notation*). W ten sposób można zastąpić obydwie standardowe metody dostępowe – decyzja o tym, która z nich zostanie wywołana (getter czy setter) jest podejmowana automatycznie w zależności od kontekstu.

Warto wiedzieć: *możliwa jest zmiana domyślnych nazw metod dostępowych. Wystarczy podać nowe nazwy przypisane odpowiednim metodom dostępowym (getter= lub setter=), np.:*

```
@property (getter=isSelected) BOOL selected;
```

c. **Atrybuty pól** – dyrektywa @property może przyjmować następujące atrybuty określające jej właściwości (m.in.):
   - **readonly** – pole wyłącznie do odczytu. Metoda setter nie jest tworzona.
   - **nonatomic** – nadpisuje domyślne ustawienie typu **atomic**. Właściwość ta związana jest z wielowątkowym przetwarzaniem, umożliwia wyłączenie automatycznego zabezpieczenia przed jednoczesnym wywołaniem metod dostępowych poprzez wiele wątków.
   - **strong** – zapewnia, że obiekt będzie istniał tak długo, jak będzie przypisany do danego pola (czyli zwiększony zostanie licznik referencji, dzięki czemu pamięć przewidziana do przechowania obiektu nie zostanie automatycznie zwolniona). Domyślna, intuicyjna prawidłowość w większości przypadków.
   - **weak** - w przypadku dwóch obiektów zawierających referencje do siebie nawzajem konieczne jest ustawienie relacji typu weak przynajmniej w jednym

polu. Standardowo przyjmuje się, że obiekt nadrzędny powinien mieć referencję typu strong, zaś obiekt podrzędny – typu weak. Atrybut weak dopuszcza możliwość usunięcia z pamięci obiektu przechowywanego w danym polu (w tym wypadku licznik referencji nie zwiększa się podczas przypisania obiektu jako wartości pola).

```
@interface Shelf: NSObject

@property (nonatomic, strong) Product *product;

@end
```

```
@interface Product : NSObject

@property (nonatomic, weak) Shelf *shelf;

@end
```

- **copy** – pozwala na tworzenie kopii obiektu w trakcie przypisywania go jako wartości danego pola (zamiast przejmowania kontroli nad obiektem, jak w przypadku atrybutu strong).

d. **Implementacja** – właściwy kod klasy, odpowiadający za jej funkcjonalności. Jest to plik o rozszerzeniu `.m` i nazwie odpowiadającej nazwie implementowanej klasy. Na początku powinien zawierać linię opowiadającą za importowanie interfejsu. Następnie przy pomocy dyrektywy `@implementation` rozpoczyna się kod klasy. Między początkowymi nawiasami klamrowymi można tu umieścić deklaracje prywatnych pól. Poza pierwszymi nawiasami znajdują się zwykle implementacje metod. Nie ma konieczności powtarzania deklaracji pól z dyrektywą `@synthesize` (od wersji Xcode 4.4+) – metody dostępowe są generowanie automatycznie bez tego dodatkowego wskazania.

```
// Product.m
#import "Product.h"

@implementation Product {
    double _sale;
}

- (void)increasePrice:(double)percentage {
    self.price = self.price * percentage/100.0;
}

@end
```

e. **Inicjalizacja** – zamiast konstruktorów w Objective-C stosowana jest inicjalizacja obiektów przy pomocy domyślnej metody `init`. Istnieje możliwość nadpisania domyślnej metody poprzez implementację dowolnej metody o nazwie rozpoczynającej się od słowa kluczowego `init`. Można w ten sposób zapewnić ustawienie wartości wybranych pól podając odpowiednie parametry metody inicjalizującej.

```objc
// Product.m
-(id)initWithPrice:(double)aPrice {
    self = [super init];
    if (self) {
        _price = aPrice;
        _sale = 0;
    }
    return self;
}
```

f. **Obiekty** – tworzenie konkretnych instancji klas (obiektów) związane jest z koniecznością wywołania dwóch metod:
   - `alloc` – pozwala na przydzielenie pamięci,
   - `init` – inicjalizuje obiekt (odpowiednik konstruktora).

   Wszystkie obiekty przechowywane są jako wskaźniki. Do wywołania metody danego obiektu należy podać w nawiasach kwadratowych jego nazwę wraz z nazwą metody, oddzieloną spacją. Aby odnieść się do pól obiektu można korzystać z ich metod dostępowych. Alternatywą jest uniwersalna notacja z operatorem kropki.

```objc
// main.m
#import <Foundation/Foundation.h>
#import "Product.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Product *product = [[Product alloc] init];
        // lub własna metoda inicjalizująca:
        // Product * product = [[Product alloc] initWithPrice:50.0];

        [product increasePrice:20.0];
        NSLog(@"Product has new price: %.1f", [product price]);
        NSLog(@"Another way to get the price value: %.1f", [product.price]);
    }
    return 0;
}
```

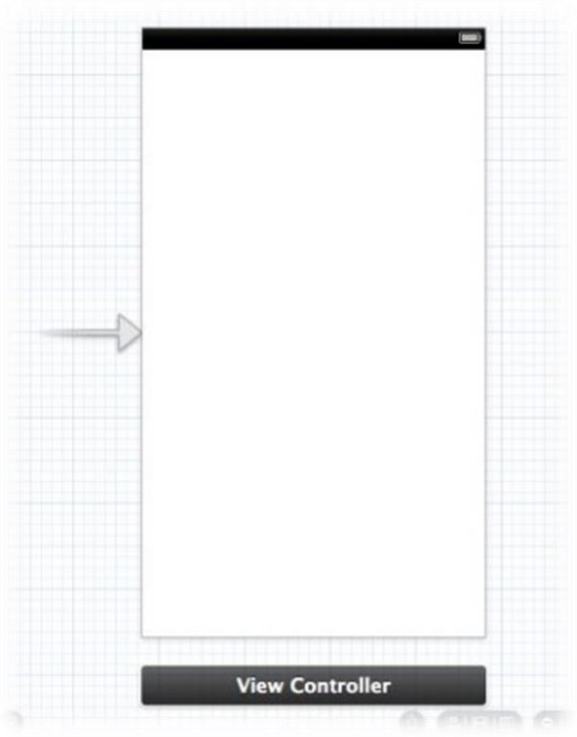
TEMAT: Programistyczny interfejs aplikacji.

WPROWADZENIE

1. Wykorzystując zdobytą wiedzę na temat języka Objective-C możesz przystąpić do implementacji pierwszej aplikacji na platformę iOS. Podobnie jak Android Studio w przypadku aplikacji dedykowanych na platformę Android, tak też środowisko Xcode w dużym stopniu wspiera programistów w tworzeniu aplikacji na platformę iOS.
2. Implementacja nowej aplikacji wiąże się z koniecznością stworzenia pliku *main.m* (przy tworzeniu nowego projektu odbywa się to automatycznie). W pliku tym framework *UIKit* tworzy w metodzie *main* podstawową funkcję *UIApplicationMain*. Funkcja ta jest wywoływana z opcją *@autoreleasepool*, która wspiera automatyczne zarządzanie czasem życia obiektów (ARC –

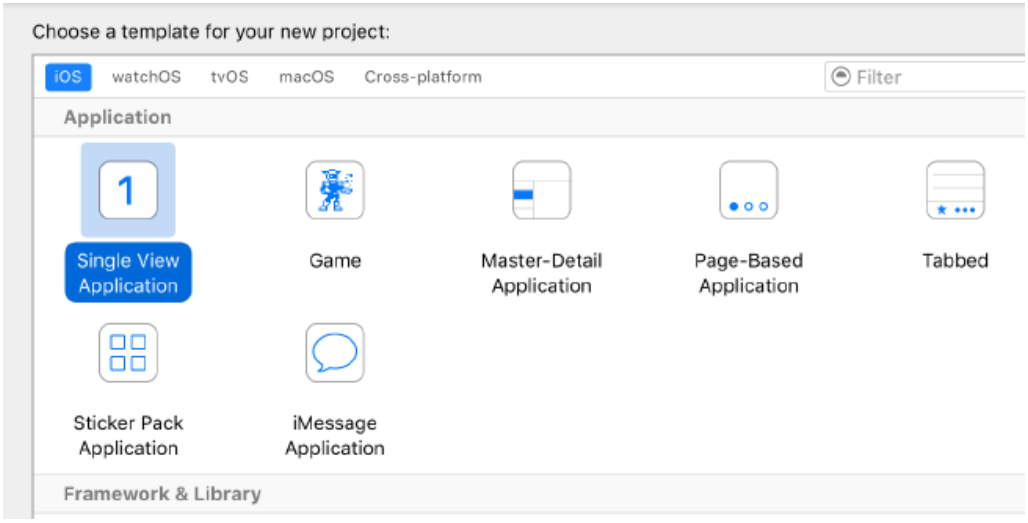
Automatic Reference Counting). W ten sposób zapewnione jest m.in. automatyczne usuwanie nieużywanych obiektów.

3. Widoki można tworzyć niezależnie od kodu metodą *drag&drop* przy pomocy wbudowanego edytora ***Interface Builder***.
4. Kompletna aplikacja składa się z wielu widoków, między którymi można nawigować. Relacje (połączenia) między widokami definiuje się za pomocą narzędzia do projektowania nazywanego ***Storyboard*** (jest on częścią edytora *Interface Builder*).
5. *Storyboard* zawiera sceny (*scenes*) oraz przejścia między scenami (*segues*). Scena stanowi reprezentację kontrolera widoku (*ViewController*).
6. Strzałka widoczna po lewej stronie sceny jest początkowym wskaźnikiem sceny, który identyfikuje główną scenę. To ten widok zostanie załadowany podczas uruchomienia aplikacji.



TREŚĆ ZADANIA

1. **Cel:** stworzenie prostej aplikacji z dwoma widokami oraz możliwością nawigacji między nimi.
2. W celu wykonania zadania należy postępować zgodnie z poniższym opisem:
   a. Stwórz nowy projekt w środowisku Xcode: wybierz szablon *Single View Application.*



   b. Kliknij przycisk *Next* na dole okna. Pojawi się kolejne okno ze szczegółami konfiguracji. Wpisz nazwę projektu i uzupełnij pozostałe wymagane pola. Jako język implementacji powinien być wybrany *Objective-C.*

c. Po kliknięciu *Next* możesz skonfigurować pozostałe ustawienia projektu, takie jak docelowa wersja oprogramowania przewidziana do działania aplikacji.
d. Edytuj plik interfejsu ViewController.h tak, aby zawierał dwa pola:
```
@property (nonatomic, weak) IBOutlet UILabel *messageLabel;
@property (nonatomic, weak) IBOutlet UITextField *inputField;
```

Zwróć uwagę na kwalifikator *IBOutlet* umieszczony przed deklaracją komponentów. IB stanowi skrót od Interface Builder. Wskazuje to na przeznaczenie tego kwalifikatora – służy on do oznaczania zmiennych, do których można odnosić się w narzędziu Interface Builder. Istnieje również analogiczny kwalifikator do oznaczania metod – *IBAction*.

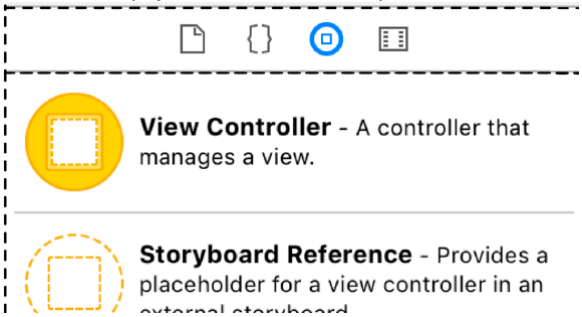
e. Metoda, która będzie wywoływana po kliknięciu przycisku powinna być zadeklarowana w pliku *ViewController.h* oraz zaimplementowana w pliku *ViewController.m*:
```
-(IBAction)enter {
}
```
f. Wewnątrz metody *enter* utwórz trzy zmienne typu *NSString*. Pierwsza z nich będzie pobierać wartość edytowalnego pola tekstowego (zmienna *inputField* typu *UITextField*) w głównym ekranie aplikacji. Zmienna *message* będzie przechowywać tekst, który zostanie wyświetlony. Zmiennej *myName* przypisz swoje imię.
```
NSString *yourName = self.inputField.text;
NSString *myName = @"YourName";
NSString *message;
```
g. Kontynuując implementację metody *enter*, dodaj zabezpieczenie przed przypadkiem, w którym użytkownik wciśnie przycisk zanim wpisze swoje imię. Poprawne działanie aplikacji w tym wypadku można zagwarantować w następujący sposób:
```
if ([yourName length] == 0) {
  yourName = @"World";
}
```
h. Następnie dopisz weryfikację, czy wpisane przez użytkownika imię jest takie same, jak Twoje. Przygotuj dwie wersje wiadomości *message* na wypadek identycznych imion oraz różnych. Do porównania dwóch łańcuchów znaków wykorzystaj metodę *isEqualToString*. Zapamiętaj, aby **nie porównywać** łańcuchów *NSString* (które są obiektami) poprzez operator *==*. W przeciwnym wypadku w większości zastosowań zwracany wynik nie będzie zgodny z oczekiwaniami. Operator == porówna adresy obiektów, nie ich wartości.
```
if ([yourName isEqualToString:myName]) {
        message = [NSString stringWithFormat:@"Hello %@! We have
the same name :)", yourName];
}else {
        message = [NSString stringWithFormat:@"Hello %@!", yourName
];
}
```
i. Kolejna linia metody *enter* będzie odpowiadała za wyświetlenie utworzonej wiadomości w polu *messageLabel* typu *Label*:
```
self.messageLabel.text = message;
```
j. Kompletna metoda *enter* powinna wyglądać w następujący sposób:

```
- (IBAction) enter {
    NSString *yourName = self.inputField.text;
    NSString *myName = @"YourName";
    NSString *message;

    if ([yourName length] == 0) {
        yourName = @"World";
    }
    if ([yourName isEqualToString:myName]) {
        message = [NSString stringWithFormat:@"Hello %@. We have the same name :)", yourName];
    } else {
        message = [NSString stringWithFormat:@"Hello %@!", yourName];
    }

    self.messageLabel.text = message;
}
```

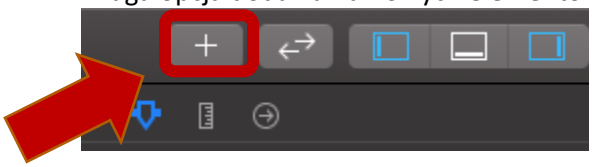
k. Plik *ViewController.m* zawiera domyślnie jeszcze metodę *viewDidLoad()*.

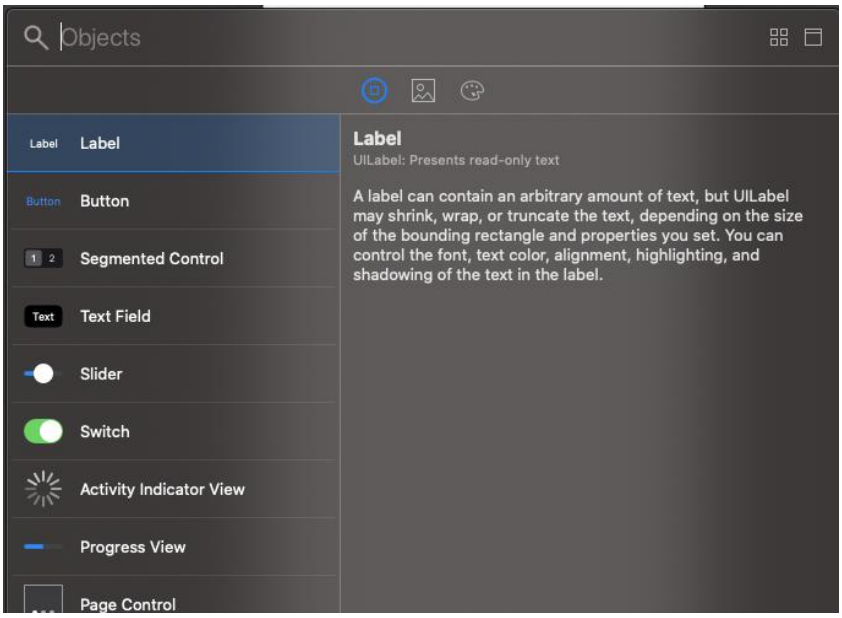
l. Po stworzeniu pól i metod kontrolera *ViewController*, związanego z pierwszym widokiem (oknem) aplikacji, możesz przystąpić do definiowania widoku w zintegrowanym narzędziu *Interface Builder*.

m. W panelu nawigacyjnym projektu (*Project Navigator*) kliknij *Main.storyboard*. Gdy obszar roboczy zostanie załadowany, dodaj niezbędne elementy widoku (wykorzystane w kontrolerze): *UIButton, UILabel, UITextField.*
Pierwsza opcja dodawania nowych elementów widoku:



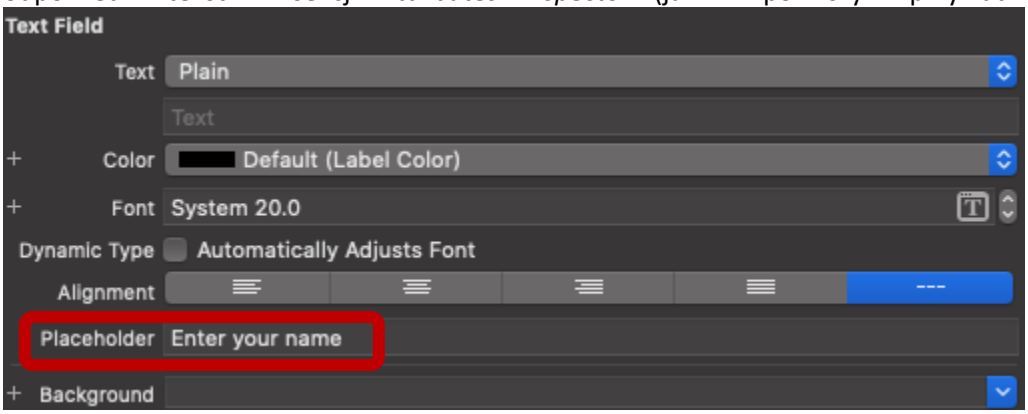
Druga opcja dodawania nowych elementów widoku:



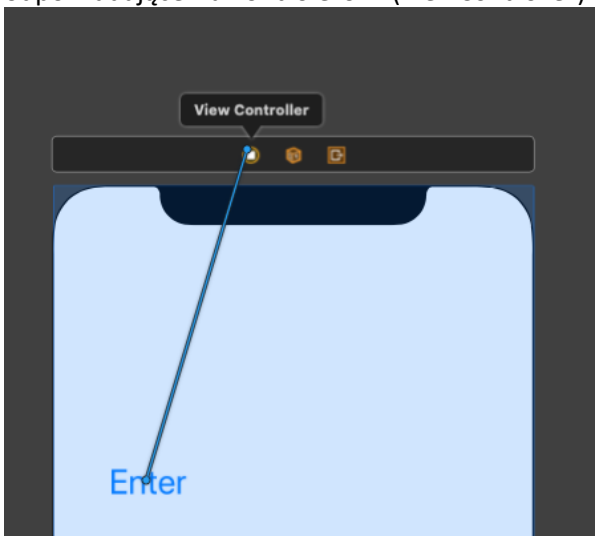


n. Poprzez dwukrotne kliknięcie na każdym z dodanych obiektów możesz zmienić ich nazwy. Zmodyfikuj ustawienia tak, aby uzyskać następujący efekt (nazwa przycisku to *Enter*, pole typu *Label* pozostaje puste, w edytowalnym polu tekstowym wyświetla się podpowiedź, co należy wpisać):

o. Aby w edytowalnym polu tekstowym wyświetlała się podpowiedź, należy wpisać odpowiedni tekst w sekcji *Attributes Inspector* (jak w poniższym przykładzie).

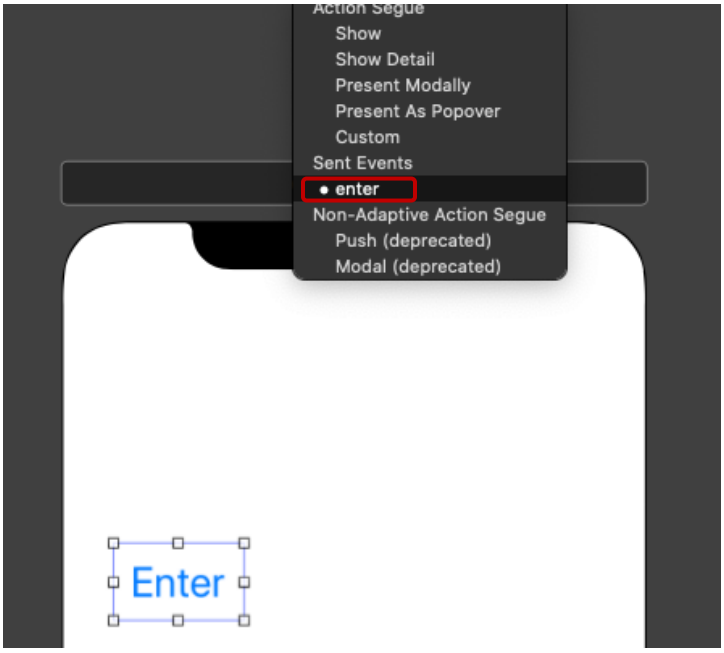


p. Dodatkowo należy zapewnić, że wpisane przez użytkownika imię będzie zawsze rozpoczynało się od wielkiej litery (w menu atrybutów pole *Capitalization -> Words*).
q. Następny krok obejmuje powiązanie elementów interfejsu użytkownika z klasą dziedziczącą po kontrolerze widoku (plik *ViewController,* w którym są już zdefiniowane odpowiednie pola i metoda).
r. Trzymając przycisk *Ctrl* należy przeciągnąć przycisk *UIButton* do przycisku odpowiadającemu kontrolerowi (*ViewController*) na górze danego *storyboard*.

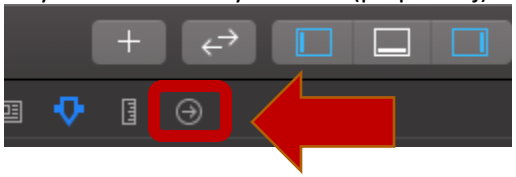


Pojawią się wtedy następujące opcje:

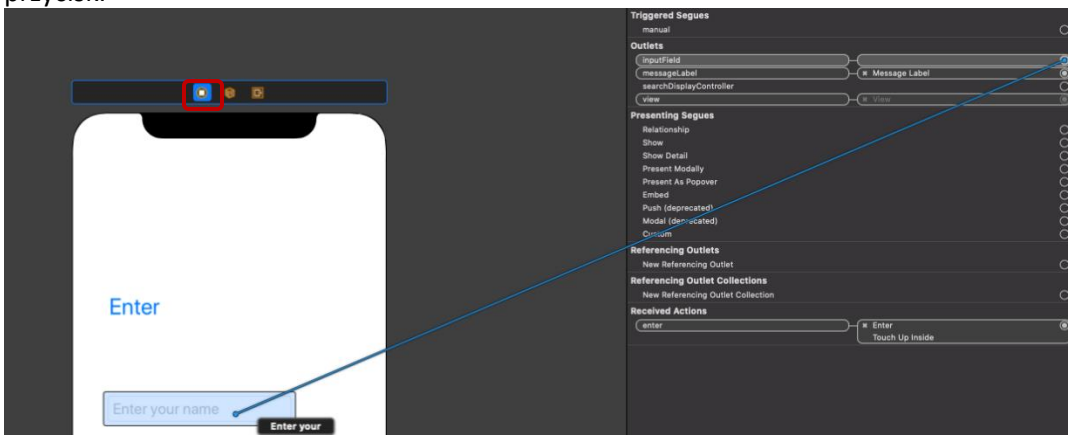
Należy wybrać *Enter*.



Inna opcja to zaznaczenie widoku okna (przycisk *View Controller*, jak w opisie powyżej) i wybranie w bocznym menu (po prawej) zakładki *Show Connections Inspector*:



W dostępnym menu widoczna będzie nazwa utworzonej metody. Należy połączyć z nią przycisk.

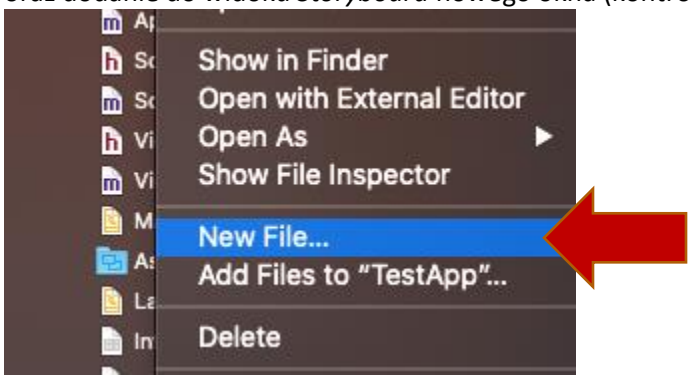


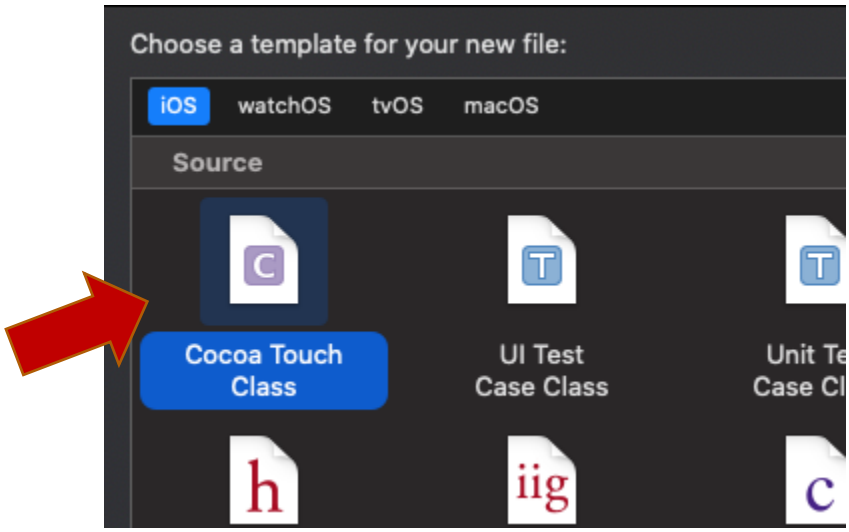
s. Podobnie należy postępować z innymi kontrolkami. Zarówno edytowalne pole tekstowe, jak i etykieta powinny zostać również połączone z implementacją (umożliwi to dynamiczne uaktualnianie tekstu).

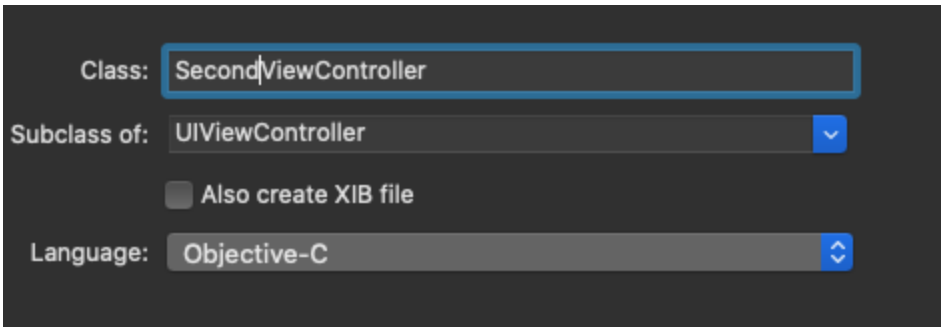
t. Sprawdź, czy aplikacja działa zgodnie z oczekiwaniami.

3. Druga część zadania obejmuje dodanie kolejnego widoku. Między widokami zostanie przesłany napis podany przez użytkownika.

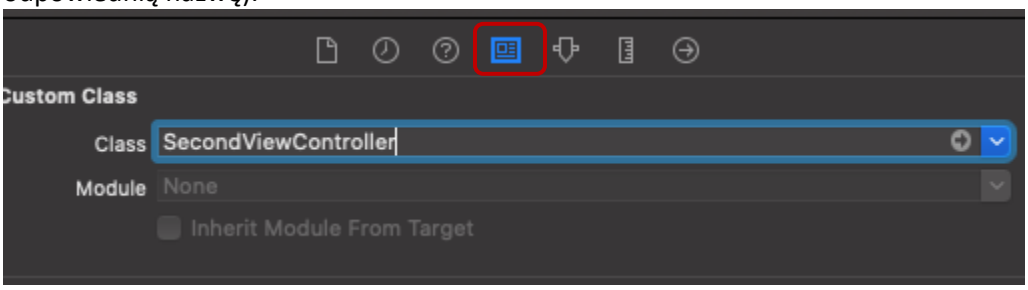
a. Konieczne będzie utworzenie klasy drugiego kontrolera (New File > Cocoa Touch Class) oraz dodanie do widoku *Storyboard* nowego okna (kontrolka typu *ViewController*).







b. Przypisz nowemu widokowi drugi kontroler (Second*ViewController*). W tym celu wybierz w edytorze *Storyboard* drugie okno (poprzez kliknięcie w jego obszarze w górnym panelu w pierwszy od lewej strony żółty przycisk o nazwie *ViewController*), a następnie w menu po prawej stronie w sekcji *Identity Inspector* wybierz z listy *Class* drugi kontroler (lub wpisz odpowiednią nazwę).



c. Dodaj do pierwszego widoku kolejny przycisk i połącz go z nowym widokiem (poprzez przeciągnięcie kursorem i wybranie opcji *Show*). Zmień ustawienia przycisku (kolor, czcionkę, wielkość itp.).

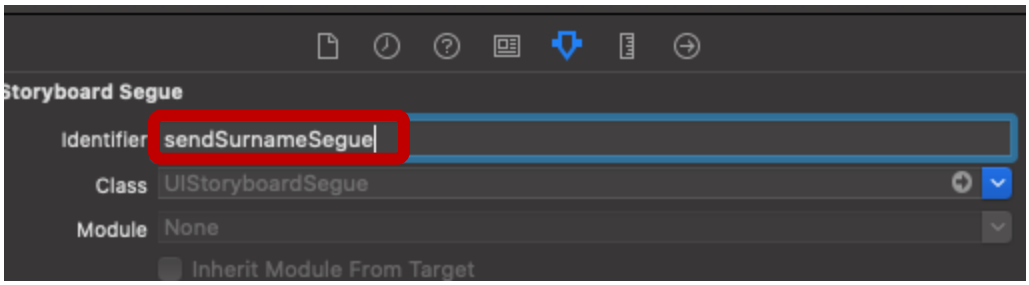
d. Dodaj także kolejne edytowalne pole tekstowe (użytkownik będzie podawał w nim nazwisko). Zadbaj o to, aby zmienna miała metody dostępowe.

e. W drugim widoku (w *Storyboard*) oraz kontrolerze (*SecondViewController*) dodaj edytowalne pole tekstowe (nazwa: *modifiedSurnameInputField*) oraz zmienną typu *NSString* do tymczasowego przechowania nazwiska (np. pole o nazwie *surname*).

```
21 @interface SecondViewController : UIViewController
22
23 @property (nonatomic, weak) IBOutlet UITextField *modifiedSurnameTextField;
24
25 @property NSString *surname;
```

f. Dodaj następujący wpis do pierwszego kontrolera (*ViewController*):
`#import "SecondViewController.h"`

g. Przejściami między widokami zarządza *Segue*. Po wywołaniu *segue*, uruchamiana jest metoda *prepareForSegue:sender:*. Zaimplementowanie tej metody pozwala na przesyłanie danych do innego kontrolera. Dobrą praktyką jest przypisanie każdemu *segue* unikalnego identyfikatora. Nadaj więc utworzonemu połączeniu nazwę, np. *sendSurnameSegue*. W tym celu należy zaznaczyć obiekt przejścia w *Storyboard*.



h. W pierwszym kontrolerze (*ViewController*) zaimplementuj metodę *prepareForSegue*:

```
-(void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([segue.identifier isEqualToString:@"sendSurnameSegue"]) {
        SecondViewController *controller = (SecondViewController *) segue.destinationViewController;
        controller.surname = self.surnameTextField.text;
```

i. Z przesłanej wartości można skorzystać w drugim widoku w metodzie *viewDidLoad* kontrolera *SecondViewController*. Należy w ramach tej metody podstawić przesłaną wartość tekstową (nazwisko) do odpowiedniej kontrolki (pola tekstowego), aby została wyświetlona na ekranie.

j. W tym momencie użytkownik ma możliwość zmodyfikowania wyświetlonego nazwiska. Aby zatwierdzić modyfikację i przesłać zaktualizowane nazwisko do pierwszego widoku konieczne jest dodanie właściwości delegata (*Delegate*) i implementacja protokołu. **Pierwszy kontroler stanie się delegatem drugiego kontrolera**. W tym celu trzeba dostosować *ViewController* (pierwszy kontroler) tak, aby spełniał wymagania narzucone w drugim kontrolerze (*SecondViewController*).

k. W pliku interfejsu drugiego kontrolera (*SecondViewController.h*) przed dyrektywą *@interface* dodaj następujący wpis:

```
@class SecondViewController;

@protocol SecondViewControllerDelegate <NSObject>

- (void) addItemViewController:(SecondViewController *) controller didFinishEnteringItem: (NSString *) item;

@end
```

l. W tym samym kontrolerze utwórz pole delegata:

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

m. Dodaj w drugim kontrolerze przycisk służący do powrotu do pierwszego widoku. Pamiętaj, aby podłączyć go do odpowiedniego pola *@property*.

n. Zaimplementuj metodę (o dowolnej nazwie) powiązaną z przyciskiem znajdującym się w widoku drugiego kontrolera:

```
NSString *itemToPassBack = self.modifiedSurnameTextField.text;
[self.delegate addItemViewController:self didFinishEnteringItem:itemToPassBack];
[self dismissViewControllerAnimated:YES completion:nil];
```

o. W pierwszym kontrolerze (w pliku nagłówkowym) dokonaj następujących zmian, które umożliwią wykorzystanie utworzonego protokołu:

```
10  #import "SecondViewController.h"
11
12  @interface ViewController : UIViewController <SecondViewControllerDelegate>
```

p. Zaimplementuj w pliku *ViewController.m* metodę zdefiniowaną w protokole:

```
- (void) addItemViewController:(SecondViewController *)controller didFinishEnteringItem:(NSString *)item {
    NSLog(@"This was returned from SecondViewController %@", item);
    self.surnameTextField.text = item;
}
```

q. Konieczne jest jeszcze jawne zadeklarowanie, że to pierwszy kontroler jest delegatem drugiego kontrolera (w metodzie *prepareForSegue* kontrolera *ViewController*).

```
controller.delegate = self;
```

r. Pamiętaj o tym, aby połączyć wszystkie obiekty widoku (np. pola tekstowe) z ich odpowiednikami w kodzie.