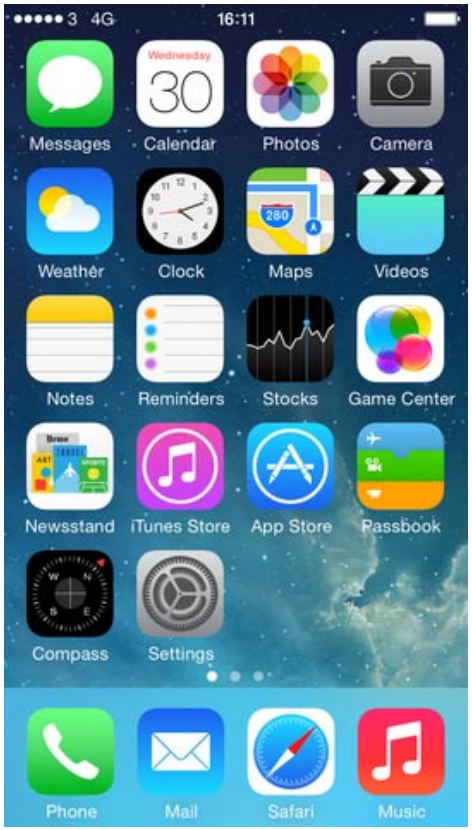
Swift- ja iOS UIKit -OHJELMOINTIHARJOITUKSIA

Kari Laitinen
http://www.naturalprogramming.com
2014-08-25 Tiedosto luotu.
2015-08-31 Harjoitukset muutettu Swift-kielelle.
2019-10-21 Otsikko uusiksi.

Syksyllä 2019 Apple julkaisi uuden teknologian nimeltä SwiftUI, jonka on tarkoitus korvata esim. UIKit iOS-sovellusten rakentamisessa.

SwiftUI-teknologiaan liittyvät harjoitukset ovat erillisessä dokumentissa.



HARJOITUKSIA OHJELMALLA Animals.swift

Näiden harjoitusten tarkoitus on opetella Swift-ohjelmointikielen rakenteita.

Animals.swift on ohjelma joka sisältää yksinkertaisen luokan nimeltä `Animal`. Ohjelmassa myös luodaan `Animal`-olioita ja kutsutaan metodeita olioiden suhteen.

Harjoitus 1:

Kun Swift-kieliseen luokkaan kirjoitetaan metodi, sen alkuun pannaan varattu sana `func`. Kääntäjä siis tunnistaa metodimäärittelyn alun tuon varatun sanan perusteella.

Lisää luokkaan `Animal` metodi `make_stomach_empty()`, jonka avulla `Animal`-olion vatsa tyhjennetään siten että sinne kirjoitetaan tyhjä stringi "". Metodia tulee voida kutsua esim. seuraavasti

```
cat_object.make_stomach_empty()
dog_object.make_stomach_empty()
```

Tämän muutoksen onnistumisen voi testata kutsumalla `make_speak()` -metodia ja tutkimalla onko vatsa todella tyhjentynyt.

Harjoitus 2:

Swift-kielen terminologiassa konstruktoreita, joilla oliot rakennetaan, kutsutaan yleisesti termillä *initializer*. Konstruktori on metodi jota kutsutaan automaattisesti silloin kun olio luodaan. Swift-kielessä konstruktorien nimenä on `init()`. Sellaisesta konstruktorista joka kutsuu luokan jotain toista konstruktoria käytetään nimitystä *convenience initializer.*

Lisää `Animal`-luokkaan ns. oletuskonstruktori jonka avulla olio voidaan luoda parametreja antamatta. `Animal`-tyypin olio voidaan luoda tämän seurauksena parametreja antamatta seuraavasti

```
var default_animal = Animal()
```

Datakentän `species_name` arvoksi voidaan oletuskonstruktorissa laittaa stringi "default animal". Oletuskonstruktori voidaan tehdä esim. käyttämällä edellä mainittua convenience initializeria. Ohjelmassa **Windows.swift** on esimerkkejä erilaisista konstruktoreista.

Tämänkin tehtävän onnistumisen voit todeta `make_speak()` -metodin avulla.

Harjoitus 3:

Luokkien datakenttiä eli datajäseniä voidaan kutsua myös instanssimuuttujiksi. Alkuperäisessä **Animals.swift**-ohjelmassa on kaksi instanssimuuttujaa `species_name` ja `stomach_contents`. Tässä tehtävässä sinun tulee listätä luokkaan `Animal` uusi datakenttä (instanssimuuttuja) nimeltä `animal_name` johon talletetaan eläimen nimi stringinä.

Tässä on tehtävä uusi konstruktori siten että `Animal`-olio voidaan luoda esim. lauseella

```
var cat_object  =  Animal( "cat", "Arnold" )
```

Swift-kielessä voidaan antaa metodien ja konstruktorien parametreille ns. eksternaaleja nimiä. Kun tämmöisiä nimiä ei haluta käyttää, kirjoitetaan parametrin paikallisen nimen eteen alaviiva. Ohjelmissa **Olympics.swift** ja **Windows.swift** on erilaisia konstruktoreita ja metodeita käytetty.

Tässä pitää myös luokan ns. kopiointikontruktoria muuttaa siten että uusi datakenttä tulee kopioiduksi. Metodia `make_speak()` tulee niin ikään modifioida siten että se tekee seuraavantapaisen tulostuksen

```
Hello, I am a cat named Arnold.
I have eaten: ...
```

Oletuskonstruktorissa voidaan uudelle datakentälle antaa oletusarvoksi "nameless".

Harjoitus 4:

Muuta metodia `make_speak()` siten että se tulostaa

```
Hello, I am a ... named ...
My stomach is empty.
```

siinä tapauksessa kun `stomach_contents` viittaa tyhjään stringiin. Vatsa on tyhjä niin kauan kuin metodia `feed()` ei ole kutsuttu. Eräs tapa tutkia että onko jokin stringi tyhjä, on käyttää `String.CharacterView`-luokan propertyä `count`. Stringin pituuden tutkimisesta on esimerkki ohjelmassa **StringReverse.swift**.

Jos vatsa ei ole tyhjä, annetaan alkuperäisen kaltainen tulostus.

HARJOITUKSIA OHJELMALLA SoccerWorldCups.swift

SoccerWorldCups.swift on esimerkkiohjelma, jonka avulla käyttäjä voi saada tietoja pelatuista jalkapallon World Cupeista eli MM-kisoista joita pelataan joka neljäs vuosi.

Tässä ohjelmassa on taulukko joka sisältää `WorldCup`-olion jokaiseen pelattuun kisaan liittyen. Tästä taulukosta haetaan tietoa sitten mm. ’filtteröimällä’ eli suodattamalla taulukossa olevia olioita tiettyjen kriteerien mukaisesti.

Harjoitus 1:

Lisää ohjelmaan tiedot tulevista tai taulukosta puuttuvista jalkapallon World Cup -kisoista. Tuleviin kisoihin voi voittajamaaksi laittaa "Not Known".

Harjoitus 2:

Nykyisellään ohjelma ei sano mitään jos sille annetaan vuosi jona jalkapallon World Cupia ei järjestetty. Muuta ohjelma sellaiseksi että se tulostaa tässä tapauksessa jotain seuraavan kaltaista:

```
"No Wold Cup was organized in ..."
```

OlympicsFiltered.swift on esimerkkiohjelma josta voi olla hyötyä tätä osatehtävää ratkaistaessa.

Harjoitus 3:

Lisää ohjelmaan menusta valittava toiminto

```
"Find World Cups hosted by a certain country"
```

Harjoitus 4:

Lisää ohjelmaan menusta valittava toiminto jolla etsitään sellaiset `WorldCup`-oliot joissa isäntämaa ja voittajamaa ovat samoja. Tällä toiminnolla voi siis etsiä kisat joissa on saattanut olla kotikenttäetua.

Harjoitus 5:

Käytä `String`-luokan `lowercased()`-metodia sopivasti siten että ohjelma osaa etsiä oikean maan nimen vaikka sille annettaisiin maan nimi pienellä alkukirjaimella.

HARJOITUKSIA OHJELMALLA SquareBallRectangle

iOS-esimerkkiohjelmissa on Swift-sovellus nimeltä SquareBallRectangle, joka näyttää neliötä, palloa, tai suorakaidetta sen mukaan mikä kuvio on `UISegmentedControl`-olion avulla valittuna.

Saat sovelluksen käyttöösi kun kopioit kurssin sivulta **SquareBallRectangle.zip**-tiedoston ja purat sen paikallisesti. Purkaminen eli unzippaus tapahtuu tiedostonimen tuplaklikkauksella tai Safari voi tehdä sen myös automaattisesti.

Varmista ensin että saat ohjelman toimimaan ennenkuin alat tekemään seuraavia harjoituksia.

Huomioi että tästä sovelluksesta on olemassa myös Objective C -pohjainen versio joka on talletettu nimellä **SquareBallRectangleOC**. Ole siis tarkkana että otat käyttöön oikean sovelluksen.

Harjoitus 1:

Muuta ohjelma sellaiseksi että se piirtää suorakaiteen (rectangle) tilalle kolmion (triangle).

Kolmion piirtämiseen ei ole valmiina metodia/funktiota, mutta kolmion saa aikaiseksi esim. kun määrittelee kolmion toteuttavan piirtopolun. Tiedoston **SquareBallRectangleView.swift** lopussa on kommenteissa valmiina ohjelmarivit, jotka määrittävät kolmion piirtämisen piirtopolun.

Kun kolmiota esittävä piirtopolku tehdään valmiiksi annetuilla koodiriveillä, kolmio piirtyy suurinpiirtein keskelle View-oliota, eikä koordinaatiston nollapistettä tarvitse tässä siirrellä millään tavalla.

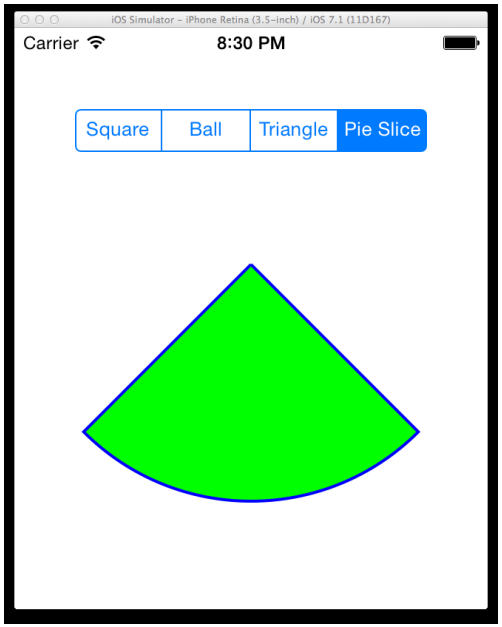
Piirtofunktioita käytettäessä piirtopolku tavallaan nollaantuu sen jälkeen kun piirto-operaatio on tehty.

Tässä osatehtävässä sinun tulee määritellä käyttöliittymään sanan "Rectangle" tilalle "Triangle".

Harjoitus 2:

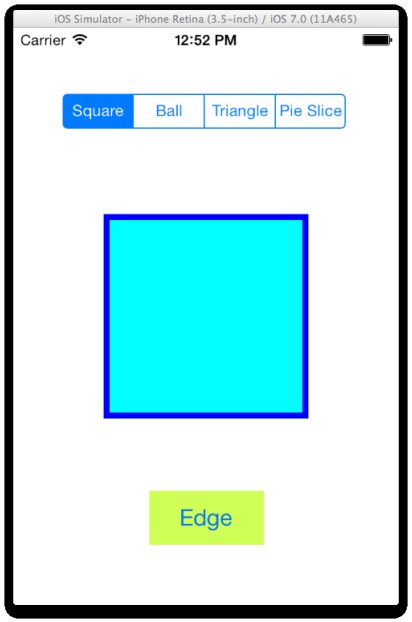
Muuta ohjelmaa siteen että valittavaksi tulee uusi kuvio nimeltä "Pie Slice", eli eräänlainen piirakkapala. Tätä varten tulee käyttöliittymän suunnittelutyökalulla muuttaa `UISegmentedControl`-olio sellaiseksi että siinä on mahdollisuus neljään valintaan. Lisäksi tulee piirtofunktio tehdä sellaiseksi että se piirtää tuon piirakan. Tässä voit katsoa mallia esim. sovelluksesta DrawingDemo.

Tässä on tarkoitus että kun valitaan "Pie Slice", ohjelma tuottaa suurinpiirtein seuraavan kuvan kaltaisen näkymän



Harjoitus 3:

Muuta ohjelmaa siten että siihen tulee painonappi (Button), jonka avulla voi säätää piirrettävien kuvioiden reunan paksuutta. Reunan paksuus voi aina nappia painettaessa kasvaa esim. kahdella (pikselillä) kunnes se saavuttaa arvon 10, minkä jälkeen se palaa arvoon 2. Käyttöliittymä voi tämän muutoksen jälkeen näyttää seuraavanlaiselta, eli siinä on tekstillä "Edge" varustettu painonappi:



Alkuperäisessä ohjelmassa piirtoviivan leveys asetetaan piirtometodin `draw()` alussa. Nyt tuo viivanleveys tulee lukea jostain datajäsenestä, joka voi olla määritelty esim. seuraavasti

```
var edge_line_width : CGFloat = 2
```


Ohjelmaan tulee määritellä metodi jota kutsutaan silloin kun nappia painetaan. Ko. metodin voi laittaa esim. `UIView`-luokasta johdetun luokan sisään. Lisäksi täytyy käyttöliittymätyökaluilla lisätä painonappi ja muodostaa yhteys painonapin ja tuon uuden metodin välille. ButtonDemo on yksinkertainen sovellus josta näet minkälainen voi olla painonappiin reagoiva metodi.

HARJOITUKSIA OHJELMALLA GesturesDemo

iOS-käyttöjärjestelmässä on luokat joiden avulla voidaan saada selville minkälaisia ’eleitä’ (gestures) käyttäjä tekee kosketusnäytön avulla. Eleitä on mm. seuraavanlaisia

- Tap: näyttöä kosketetaan
- Swipe: jonkinlainen pyyhkäisy, ei toimine iOS-simulaattorissa
- Long Press: näyttöä kosketetaan pidempään
- Pan: näyttöä pyyhkäistään, saadaan aikaan simulaattorissa
- Pinch: kahdella sormella skaalataan, voidaan simuloida
- Rotation: kahdella sormella pyöritetään, voidaan simuloida

GesturesDemo on ohjelma jonka avulla voidaan tutkia näitä eletapahtumia ja nähdään milloin niitä generoidaan.

Tutki ja koekäytä ensin tätä ohjelmaa ja tee sitten seuraavia harjoituksia. Voit säilyttää ohjelman alkuperäiset ominaisuudet sillä niistä voi olla apua kun lisäät uusia ominaisuuksia ohjelmaan.

Harjoitus 1:

Muuta ohjelma sellaiseksi, näytöllä näytetään palloa joka on `Ball`-luokan olio.

Voit kopioida `Ball`-luokan sovelluksesta MovingBallsWithFinger esim. kun menettelet seuraavasti:

Kopioi ensin myös sovellus MovingBallsWithFinger netistä siten että sen **.zip**-tiedosto purkaantuu paikallisiksi tiedostoiksi.

Tämän jälkeen menettelet Xcodessa GesturesDemo-projektissa siten että valitset toiminnon `File -> Add Files to "GesturesDemo" ...` Kun sinua pyydetään valitsemaan tiedosto dialogissa, etsi **Ball.swift**-tiedosto MovingBallsWithFinger-sovelluksen tiedostojen joukosta ja lisää se projektiisi.

Kun sinulla on `Ball`-luokka käytössä voit laittaa ohjelmaan tiedostoon **GesturesDemoView.swift** datajäsenen

```
var ball_on_screen : Ball! = nil
```

ja sitten voit **GesturesDemoView.swift** -tiedoston piirtometodissa luoda `Ball`-olion seuraavasti

```
if ball_on_screen == nil
{
   ball_on_screen =
              Ball( 100, 220, UIColor.red.cgColor )
}
```

Kun olion luonti tehdään näin piirtometodissa, ei tarvitse miettiä minkälainen konstruktori luokkaan tarvitaan. Kun `Ball`-olio on luotu voit piirtää pallon piirtometodissa seuraavasti

```
let context: CGContext = UIGraphicsGetCurrentContext()!

ball_on_screen.draw( context )
```

Harjoitus 2:

Kun olet saanut edellisessä tehtävässä pallon ilmestymään näytölle, on tehtäväsi muuttaa ohjelma sellaiseksi että kun näytöllä tehdään Tap-operaatio, pallo siirtyy siihen paikkaan jota täpättiin.

Ohjelmassa on jo valmis metodi joka reagoi Tap-operaatioihin. Siihen on lisättävä pallon paikan muuttaminen. `Ball`-luokassa on tätä varten jo olemassa metodi. Täppäyksen paikka saadaan selville esim. seuraavasti.

```
let tapped_point = recognizer.location( in: self )
```

Ohjelmaa MovingBallsWithFinger tutkimalla saat selville kuinka `CGPoint`-oliosta kaivetaan koordinaatit esille.

Harjoitus 3:

Tässä tehtävässä ohjelma tulee muuttaa sellaiseksi että aina kun näyttöä 'täpätään', tuohon täpättyyn kohtaan ilmestyy uusi pallo. `Ball`-olio siis luodaan täppäykseen reagoivassa metodissa. `Ball`-oliot kannattaa tallettaa taulukkoon joka voidaan määritellä datajäseneksi `GesturesDemoView`-luokkaan seuraavasti

```
var balls_on_screen : [ Ball ] = [ ]
```

Tällainen taulukon määrittely tarkoittaa että tehdään muuttuva taulukko johon voidaan tallettaa `Ball`-tyyppisiä olioita ja alussa taulukko on tyhjä.

Yksittäinen `Ball`-olio voidaan lisätä taulukon loppuun `append()`-metodin avulla. Tässä pitää muuttaa `tap_detected()`-metodia siten että pallon siirtämisen sijaan siellä luodaan uusi `Ball`-olio ja lisätään se taulukkoon.

Taulukossa olevat `Ball`-oliot voidaan kaikki piirtää seuraavanlaisen `for-in` -silmukan avulla:

```
for  ball in balls_on_screen
{
   ball.draw( context )
}
```

Harjoitus 4:

Muuta ohjelma sellaiseksi että palloilla on eri väri. Voit tietysti yrittää tässä luoda satunnaisia värejä. Toinen tapa on että määrittelet datajäseneksi seuraavanlaisen taulukon

```
var ball_colors : [ UIColor ] =

   [ UIColor.red, UIColor.green, UIColor.yellow,
     UIColor.cyan, UIColor.magenta, UIColor.blue,
     UIColor.brown, UIColor.gray, UIColor.orange ]
```

Kun tämmöinen taulukko on käytössä, voit ottaa sieltä esim. viimeisen värin ja siirtää sen sitten taulukkoon ensimmäiseksi. `UIColor`-tyyppisen värin voit ottaa taulukon lopusta ja siirtää ensimmäiseksi esim. lauseilla

```
let color_for_new_ball = ball_colors.removeLast()

ball_colors.insert( color_for_new_ball, at: 0 )
```

Tällä tavalla sama väri tulee uudelleen käyttöön vasta kun kaikki taulukon värit on kertaalleen käytetty. `UIColor`-tyyppinen väri tulee muuntaa vielä `CGColor`-tyyppiseksi ennenkuin `Ball`-olio voidaan luoda.

Harjoitus 5:

Paranna ohjelmaa siten että Pan-eleen avulla voidaan ottaa näytöltä pallo pois jos tuo Pan-ele sattuu pallon kohdalle. Tässä voidaan menetellä siten että otetaan ensin selville Pan-eleen piste seuraavanlaisella lauseella

```
let pan_point = recognizer.location( in: self )
```

Tämän jälkeen voidaan tutkia pallotaulukkoa lopusta alkaen ja tuhota sieltä ensimmäinen `Ball`-olio joka sisältään tuon pisteen. Kun pallotaulukko käydään läpi lopusta alkaen, tuhotuksi tulee nimenomaan päällimmäisin pallo. Käytännössä riittää kun tuhotaan aina vain yksi pallo koska näitä Pan-eleitä syntyy runsaasti kun näyttöä pyyhkäistään.

Tuhottavan pallon etsintään voidaan käyttää seuraavanlaista silmukkaa:

```
var ball_index = balls_on_screen.count
var ball_to_delete_is_found = false

while  ball_index > 0 && ball_to_delete_is_found == false
{
   ball_index -= 1

   if  balls_on_screen[ ball_index ].contains_point( pan_point )
   {
      // Tässä kohden täytyy pallo-olio tuhota taulukosta

      ball_to_delete_is_found = true
   }
}
```

Harjoitus 6:

Muuta ohjelma sellaiseksi että Pinch-eleellä voidaan kaikkien pallojen kokoa muuttaa suuremmaksi tai pienemmäksi. `UIPinchGestureRecognizer`-olioilla on property nimeltä `scale` jota voidaan tässä tutkia.

Tämä `scale`-arvo on suurempi kuin yksi kun sormia etäännytetään Pinch-eleessä ja se on pienempi kuin yksi kun sormia tuodaan lähemmäksi toisistaan. Tuon arvon perusteella saadaan tieto siitä pitääkö palloja suurentaa vai pienentää. `Ball`-luokassa on valmiina metodit `enlarge()` ja `shrink()` joiden avulla palloa voidaan suurentaa ja pienentää. Voit siis yksinkertaisesti käydä silmukassa pallot läpi ja suurentaa tai pienentää niitä riippuen siitä minkälainen Pinch-ele havaittiin.

Pinch-ele saadaan aikaiseksi iOS-simulaattorissa kun käytetään Alt-näppäintä ja ’hiirtä’ yhtäaikaisesti.

HARJOITUKSIA iOS-SOVELLUKSELLA ChangingViewsNoIB

Harjoitus 1:

Lisää uusi `UIButton` viewiin eli näkymään jota kontrolloi `MainViewController`. Uuden painonapin voi laittaa olemassaolevan painonapin alle. Uudella napilla voi olla sama koko kuin olemassaolevallakin painonapilla. Painonapin voi määritellä seuraavasti:

```
var button_demo_button : UIButton!
```

Tämä nimi on sopiva koska seuraavassa harjoituksessa on tarkoitus että painonapilla saa näkyville näkymän joka on käytössä sovelluksessa ButtonDemoNoIB.

Kopioimalla olemassaolevan painonapin määrittelevät koodirivit ja muuttamalla niitä sopivasti saat uuden painonapin ominaisuudet helposti tehtyä. Käytä uusia värejä uudelle painonapille. Näin aluksi uusi painonappi voi aktivoida saman viewin kuin olemassaoleva painonappi.

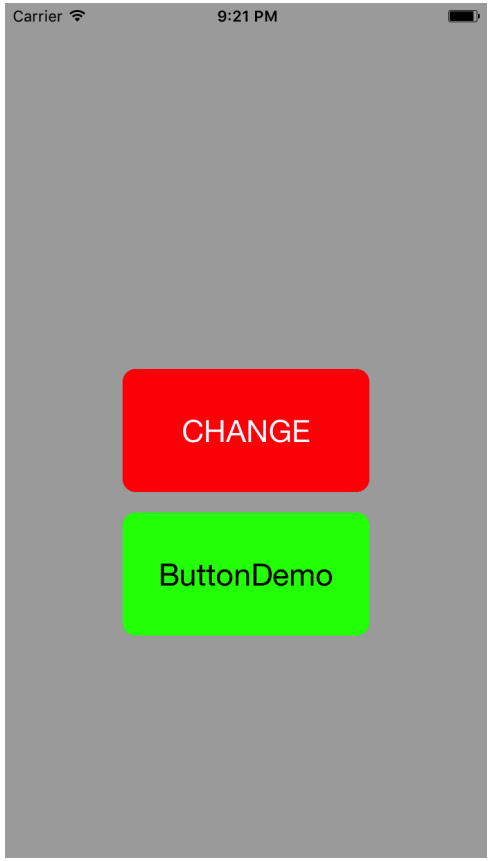
Seuraavilla constraint-määrittelyillä uusi painonappi saadaan sijoitetuksi täsmälleen vanhan painonapin alapuolelle:

```
button_demo_button.centerXAnchor.constraint(
       equalTo: view.layoutMarginsGuide.centerXAnchor ).isActive = true

button_demo_button.centerYAnchor.constraint(
```

```
equalTo: view.layoutMarginsGuide.centerYAnchor,
constant: 112 ).isActive = true
```

Näiden muutosten jälkeen `MainViewController` voisi tuottaa seuraavan näkymän:



Harjoitus 2:

Nyt tehtävänäsi on laittaa uusi painonappi aktivoimaan `ButtonDemoViewController`in joka on käytössä sovelluksessa ButtonDemoNoIB. Poimi ensin tämä sovellus käyttöösi ja pura **.zip**-tiedosto paikallisesti. Valitse sitten Xcodessa **File -> Add files to "ChangingViewsNoIB" ...**, ja lisää tiedostot **ButtonDemoViewController.swift** ja **ButtonDemoView.swift** projektiisi.

`MainViewController`-luokassa voit käyttää seuraavaa datajäsentä:

```
let button_demo_view_controller = ButtonDemoViewController()
```

Sinun pitää nyt kirjoittaa uusi metodi joka pistää tämän view controllerin toimintaan, eli ’presentoi’ sen. Niin ikään tulee uuden painonapin kyetä aktivoimaan tämä uusi metodi.

Aluksi tiedostoja **ButtonDemoViewController.swift** ja **ButtonDemoView.swift** ei tarvitse muutaa. `ButtonDemoView` pitäisi tulla näkyville kun uutta painonappia painetaan. Kuitenkin, jotta `MainViewController` saadaan takaisin näkyville, pitää tehdä muutoksia.

Ota mallia luokasta `AnotherViewController` ja lisää Back-painonappi `ButtonDemoView`-näkymään jotta päänäkymään päästään takaisin.

Harjoitus 3:

Lisää ButtonDemo-painonappi myös näkymään jota kontrolloi `AnotherViewController`. Luokkia `ButtonDemoView` ja `ButtonDemoViewController` ei tarvitse muuttaa tässä osatehtävässä.

Tämän muutoksen jälkeen pitäisi olla mahdollista että `MainViewController`ista aktivoidaan `AnotherViewController`, ja sieltä edelleen voidaan aktivoida `ButtonDemoViewController`, ja takaisin päänäkymään voidaan palata samaa ’reittiä’.