Kata Introduction

Katas are meant to be small modular exercises to practice a certain skill.
These TDD katas are divided into 10 exercises to test and develop various stages in the clean swift cycle.
These katas assume knowledge and experience with writing unit tests and are meant as exercises to strengthen the TDD muscle and so will not start with the absolute basics.
They focus on TDD specifically applied to the Clean Swift architecture.

The starting point is a common situation. You need to develop a feature but the UI hasn't been decided on yet and the backend isn't ready.
Fortunately we know the structure of the json so we can mock it and establish the models.

***A few rules:**
1. Don't change the tests in any way for the first 6 Katas.
2. Don't write more code than is necessary to make the test pass, but also think about the future and what a real project needs.
3. With that in mind don't write variables with the initial value of true in the spies.
4. Don't write mock data. The data you need is provided in Networker. Use it as needed.

Overview:

Kata 1 deals with creating the unit test Scene and then creating the true Scene for the feature.

For the Katas 2 - 6 students will be given a unit test which they need to make pass.

Kata 1 deals with creating the unit test Scene and then creating the true Scene for the feature.
Kata 2 deals with creating the test and functionality for the View Controller to the Interactor.
Kata 3 deals with creating the test and functionality for the Worker.
Kata 4 deals with creating the test and functionality for the Interactor to the Worker.
Kata 5 deals with creating the test and functionality for the Interactor to the Presenter.
Kata 6 deals with creating the test and functionality for the Presenter to the View Controller.

From Kata 7 the students write the tests and resolve the TDD cycle.

Kata 7 deals with creating the test and functionality for the ViewController to Router.
Kata 8 students implement an entire TDD cycle for "showLoadingView" and "hideLoadingView" to improve functionality
Kata 9 students implement an entire TDD cycle for "showInternetErrorView", "hideInternetErrorView", "showServiceErrorView" and "hideServiceErrorView" to improve functionality.
Kata 10 students implement the UI.

Kata 1 - SetUp the tests and scene

Below is a bit of a guide in how to start from scratch.
Optionally, if you don't have access to the Clean Swift templates or my starter files you may clone the starter project in my repo here:
https://github.com/kevinOlivet/TDDStarter.git

In the event of using the TDDStarter project in the repo skip to Kata 2.

The first Kata deals with setting up the entire process.

1. Begin by creating a folder for the scene to test and then the unit tests using the Clean Swift template. I called mine "MyFeature" and in this tutorial you should too.

If you aren't using my "Lean Unit Tests" and "Lean Scene" templates the Clean Swift template includes a lot of things that we don't need and also doesn't have many things we need.
Specifically we don't need all their sample code in the protocols and sample tests.

2. Erase the non necessary code.

We need all the spies.

3. Implement the missing spies.

4. Move the initiation of the spies to setUp and set them to nil in the tearDown.

There will be many warnings because the actual Scene doesn't exist yet. This represents the Red stage.

5. Create the folder and the actual scene in the appropriate place.

The template also comes with lots of unnecessary code meant to be a guide.

6. Erase all that nonsense.

7. In the storyboard set the ViewController to the right name in the class inspector
8. Set the storyboard id
9. Delete the original ViewController

These files are included in the project: Movement.swift, Networker.swift and GET_movements_200.json files.
***Finally replace MyFeatureModels with this:**

```
enum MyFeature {
   enum Something {
      struct Request {}
      enum Response {
         struct Success {
            var movementsResponse: [Movement]
         }
         struct Failure {
            let error: Error
         }
      }
      enum ViewModel {
         struct Success {
            var movementsViewModels: [IndividualMovement]
         }
         struct Failure {
            let title: String
            let subTitle: String
         }
         struct IndividualMovement {
            let name: String
            let amount: String
```

```
        }
      }
    }
}
```

Congratulations!
Kata 1 is finished!

Kata 2 - ViewController to Interactor

Finally it’s time to begin writing tests!

Everything begins and ends with the View Controller, specifically ViewDidLoad so we will start there too.

Typical methods are:

fetchSomething()
displaySomething(viewModel:)

showLoadingView(viewModel:)
hideLoadingView(viewModel:)

showInternetErrorView(viewModel:)
hideInternetErrorView(viewModel:)

showServiceErrorView(viewModel:)
hideServiceErrorView(viewModel:)

goToSomewhere()

For now we will focus on the core functionality: fetchMovements()
fetchMovements() should be called when the view is loaded so lets write the test:

*In MyFeatureViewControllerTests:

```
// Kata 2
func testViewDidLoad() {
    // Given
    // When
    // Then

    XCTAssertNotNil(sut, "viewDidLoad should instantiate the sut")
    XCTAssertTrue(spyInteractor.fetchMovementsCalled, "fetchMovements should be called in ViewDidLoad and call interactor fetchMovements")
  }

// Begin the Kata.  Good luck! If you need help see the hints see below.
```

Red - errors because there the BusinessLogic doesn't have fetchMovements(request:)
*In protocol MyFeatureBusinessLogic in MyFeatureInteractor:

```
// Kata 2
   func fetchMovements(request: MyFeature.Something.Request)
```

More errors because the MyFeatureInteractor doesn't conform to the protocol
*In the MyFeatureInteractor:

```
// Kata 2
   func fetchMovements(request: MyFeature.Something.Request) {
   }
```

*in MyFeatureBusinessLogicSpy in MyFeatureViewControllerTests:

```
// Kata 2
      var fetchMovementsCalled = false
      var fetchMovementsRequest: MyFeature.Something.Request?
      func fetchMovements(request: MyFeature.Something.Request) {
         fetchMovementsCalled = true
         fetchMovementsRequest = request
      }
```

No more compile time errors but the test won't pass! We need to call it.
*In MyFeatureViewController:

```
// Kata 2
   func fetchMovements() {
      let request = MyFeature.Something.Request()
      interactor?.fetchMovements(request: request)
   }
```

***Add fetchMovements() to viewDidLoad()**

Run again and it's all good.

Where's the code Coverage?
Set it up in the scheme.
Don't know how? Google it.

Congratulations! You finished Kata 2. Easy eh! Don't worry, it'll get harder.

Kata 3 MyFeatureWorker

Pretty easy so far eh? Ok, now we need to increase the difficulty a little.
The backend STILL isn't ready so we need to mock some things.
Fortunately we know what the json will look.

Let's test a successful call.
*In MyFeatureWorkerTests

```
// Kata 3
  func testFetchMovementsSuccess() {
    let url = "https://www.apple.com/mac/"

    let urlPath = URL(string: url)?.path
    // Given
    let movementsFile = "GET_movements_200.json"
    stub(condition: isPath(urlPath!)) { _ in
      let stubPath = OHPathForFile(movementsFile, type(of: self))
      return fixture(
        filePath: stubPath!,
        status: 200,
        headers: ["Content-Type" as NSObject: "application/json" as AnyObject]
      )
    }

    let movementExpectation = self.expectation(description: "calls the callback with a resource object")

    // When

    sut.fetchMovements { (movements, error) in
      // Then
      XCTAssertNotNil(movements, "successful return should have movements")
      XCTAssertNil(error, "successful return should not have an error")
      XCTAssertEqual(movements?.first?.name, "Kevin", "name should match json")
      XCTAssertEqual(movements?.first?.amount, 123, "name should match json")
      XCTAssertEqual(movements?.last?.name, "Julian", "name should match json")
      XCTAssertEqual(movements?.last?.amount, 9012, "name should match json")
      movementExpectation.fulfill()
    }

    self.waitForExpectations(timeout: 0.3, handler: nil)
    OHHTTPStubs.removeAllStubs()
  }


// Begin the Kata.  Good luck! If you need help see the hints see below.
```

Now we are going to use the OHHTTPStubs framework to intercept the call to the internet.
We could mock the Networker but we want to get closer to the true functionality with these tests so we'll use stubs instead.
Also it will keep the tests separated so the Networker tests don't grow super super long.

First we need to install the framework.

Close the project
Next make a podFile and add this to the target for tests.
pod 'OHHTTPStubs/Swift'
pod install and start using the workspace

The way it works is if the url meets the condition of being the url we specify it will return the .json file we tell it.
That way we can continue testing and developing while the backend isn't ready.
I'll just use "https://www.apple.com/mac/" until they get their act together.

Whew! Ready for some WorkerTests!
*In MyFeatureWorkerTests
import OHHTTPStubs

Run tests so it knows about OHHTTPStubs if it doesn't already.

Time to fill out MyFeatureWorker
*In MyFeatureWorker

```
// Kata 3
var networker: NetworkerProtocol = Networker()
```

*In fetchMovements(completionHandler:) in MyFeatureWorker

```
func fetchMovements(completionHandler: @escaping ([Movement]?, Error?) -> Void) {
    networker.getMovimientos(successCompletion: { (movements) in
        completionHandler(movements, nil)
    }) { (error) in
        completionHandler(nil, error)
    }
}
```

Congratulations! You've finished Kata 3!

Extra credit.
Test for the failure! It's super easy using the stubs.

*In WorkerTests

```
// Kata 3.5
func testFetchMovementsFail() {
    let url = "https://www.apple.com/mac/"

    let urlPath = URL(string: url)?.path
    // Given
    stub(condition: isPath(urlPath!)) { _ in
        let notConnectedError = NSError(domain: NSURLErrorDomain, code:
URLError.notConnectedToInternet.rawValue)
        return OHHTTPStubsResponse(error: notConnectedError)
    }
    let movementExpectation = self.expectation(description: "down network")

    // When
    sut.fetchMovements { (movements, error) in
        // Then
```

```
            XCTAssertNil(movements, "fail should not have movements")
            XCTAssertNotNil(error, "fail should have an error")
            movementExpectation.fulfill()
        }

        self.waitForExpectations(timeout: 0.3, handler: nil)
        OHHTTPStubs.removeAllStubs()
    }
```

Congratulations! You finished Kata 3

Kata 4 - MyFeatureInteractor to MyFeatureWorker

Remember I told you that the backend had already has the json established?
So fortunately we already have the model in Movement.swift
If you didn't then now is when you would have to define it.

Time for the next test.
*In MyFeatureInteractorTests

```
func testFetchMovements() {
    // Given
    let request = MyFeature.Something.Request()
    // When
    sut.fetchMovements(request: request)
    // Then

    // Kata 4
    XCTAssertTrue(spyWorker.fetchMovementsCalled, "fetchMovements should call the worker fetchMovements")
}
```

// Begin the Kata. Good luck! If you need help see the hints see below.

The MyFeatureWorkerSpy will complain. So let’s override fetchMovements and give it a sample movement to return.
*In MyFeatureWorkerSpy in MyFeatureInteractorTests

```
// Kata 4
    var fetchMovementsCalled = false
    override func fetchMovements(completionHandler: @escaping ([Movement]?, Error?) -> Void) {
      completionHandler([], nil)
      fetchMovementsCalled = true
    }
```

No more errors but the test doesn’t pass because we didn’t call it yet.
*In MyFeatureInteractor in fetchMovements(request:)

```
// Kata 4
worker.fetchMovements { (movements, error) in
}
```

Congratulations! You finished Kata 4!

Kata 5 - MyFeatureInteractor to MyFeaturePresenter

Still in MyFeatureInteractor we will need to send the information to MyFeaturePresenter for formatting.
*In MyFeatureInteractorTests in func testFetchMovements() add a further test
// Kata 5
XCTAssertTrue(spyPresenter.presentMovementsCalled, "fetchMovements should call the presenter presentMovements")

// Begin the Kata. Good luck! If you need help see the hints see below.

Red - The compiler will automatically complain that the presenter doesn't have presentMovements

*In protocol MyFeaturePresentationLogic in MyFeaturePresenter

```
// Kata 5
   func presentMovements(response: MyFeature.Something.Response.Success)
```

Now MyFeaturePresenter doesn't comply with the protocol

*In MyFeaturePresenter

```
// Kata 5
   func presentMovements(response: MyFeature.Something.Response.Success) {
   }
```

And to make MyFeaturePresenter comply with the protocol

*In MyFeaturePresenter

```
// Kata 5
   func presentMovements(response: MyFeature.Something.Response.Success) {
   }
```

And to make the MyFeaturePresentationLogicSpy comply

*In MyFeaturePresentationLogicSpy in MyFeatureInteractorTests

```
// Kata 5
      var presentMovementsCalled = false
      var presentMovementResponse: MyFeature.Something.Response.Success?
      func presentMovements(response: MyFeature.Something.Response.Success) {
         presentMovementsCalled = true
         presentMovementResponse = response
      }
```

No more errors but the test doesn't pass because we didn't call it yet.

*In worker.fetchMovements() in fetchMovements(request:) in MyFeatureInteractor

```
// Kata 5
         guard let movements = movements else { return }
         let response = MyFeature.Something.Response.Success(movementsResponse: movements)
         self.presenter?.presentMovements(response: response)
```

Now everything passes! Hurrah!

Congratulations!! You've finished Kata 5!

Kata 6 - MyFeaturePresenter to MyFeatureView Controller

Let’s take a moment to complete the VIP cycle.
We don’t have the data yet from the Worker but thanks to the modularity of Clean Swift we don’t even need it.
MyFeaturePresenter will present something and that’s all it knows. We just pass it some fake info.

So let’s test it.
*In MyFeaturePresenterTests

```
// Kata 6
   func testDisplayMovements() {
      // Given
      let movement = Movement(name: "testName", amount: 1234.56)
      let response = MyFeature.Something.Response.Success(movementsResponse: [movement])
      // When
      sut.presentMovements(response: response)
      // Then
      XCTAssertTrue(spyViewController.displayMovementsCalled, "presentMovements should display Movements in the viewController")
      XCTAssertEqual(spyViewController.displayMovementsViewModel?.movementsViewModels.first?.name, "Name: testName", "presentMovements should change the name to the correct format")
      XCTAssertEqual(spyViewController.displayMovementsViewModel?.movementsViewModels.first?.amount, "Amount: $1234.56", "presentMovements should change the amount to the correct format")
   }
```

If you aren’t used to Clean Swift then maybe you haven’t been doing the parameters correctly.
If so, then you’ll probably need to refactor to get it to compile with the response parameter.
In that case I suggest you comment out the 2 XCTAssertEqual tests and tackle the first one then move on to the second one.
If you really have problems you can check out the previous Kata’s solutions.

// Begin the Kata. Good luck! If you need help see the hints see below.

Immediately the compiler complains that there is no displayMovements
*In MyFeatureDisplayLogic in there MyFeatureViewController
func displayMovements(viewModel: MyFeature.Something.ViewModel.Success)

The compiler complains that MyFeatureViewController doesn't comply to MyFeatureDisplayLogic
*In MyFeatureViewController

```
// Kata 6
   func displayMovements(viewModel: MyFeature.Something.ViewModel.Success) {
   }
```

Now the MyFeatureDisplayLogicSpy needs to comply to MyFeatureDisplayLogic
*In MyFeatureDisplayLogicSpy in MyFeaturePresenterTests

```
// Kata 6
      var displayMovementsCalled = false
      var displayMovementsViewModel: MyFeature.Something.ViewModel.Success?
      func displayMovements(viewModel: MyFeature.Something.ViewModel.Success) {
         displayMovementsCalled = true
         displayMovementsViewModel = viewModel
      }
```

Now it compiles but the test will fail because we need to call it
*In presentMovements(response:) in MyFeaturePresenter

```
// Kata 6
      var movementArray: [MyFeature.Something.ViewModel.IndividualMovement] = []
      for movement in response.movementsResponse {
         let vm = MyFeature.Something.ViewModel.IndividualMovement(name: "Name: \(movement.name)",
amount: "Amount: $\(movement.amount)")
         movementArray.append(vm)
      }
      let viewModel = MyFeature.Something.ViewModel.Success(movementsViewModels: movementArray)
      viewController?.displayMovements(viewModel: viewModel)
```

Congratulations! You've finished Kata 6!

Kata 7 - Entire TDD cycle for loadingViews

The training wheels are off. Time for you to fly little bird. Write your own tests as you see fit.

The UX team wants to show a loading view when fetchMovements() begins and hide it when call returns.
Implement the entire TDD cycle to display a LoadingView and hide a LoadingView to improve functionality.

The loading view will have a title and a subtitle.
The title will say “Hi”
The subtitle will say “Wait a moment please”

Test the methods individually and as part of the fetchMovements() test.
You’ll need to implement or refactor fetchMovements() and it’s tests in some way.

Don’t forget to use the viewModels.
Have fun!

// Begin the Kata. Good luck! If you need help see the hints see below.

Begin with the tests. I began in MyFeatureInteractorTests, although you might begin in MyViewControllerTests if you believe the ViewController should have this functionality.
Now we are going to do both the loading and hiding at the same time to save space and prevent repetition.

*In MyFeatureInteractorTests

```
// Kata 7
   func testShowLoadingViewIsCalledWhenFetchMovementsIsCalled() {
      // Given
      let request = MyFeature.Something.Request() // When
      // When
      sut.fetchMovements(request: request)

      // Then
      XCTAssertTrue(spyPresenter.presentLoadingViewIsCalled, "fetchMovements should call the presenter showLoadingView")
   }
   // Kata 7
   func testHideLoadingViewIsCalledWhenFetchMovementsIsCalled() {
      // Given
      let request = MyFeature.Something.Request()
      // When
      sut.fetchMovements(request: request)

      // Then
      XCTAssertTrue(spyPresenter.hideLoadingViewIsCalled, "fetchMovements should call the presenter hideLoadingView")
   }


// Try to solve it with these tests.  If you need more help see below.
```

First we need to make the function.
*In MyFeatureInteractor

```
// Kata 7
   func presentLoadingView() {
      let response = MyFeature.LoadingView.Response(title: "Hi", subTitle: "Wait a moment")
      presenter?.presentLoadingView(response: response)
   }

   func hideLoadingView() {
      presenter?.hideLoadingView()
   }
```

The first error seems that MyFeature has no member LoadingView.
*In MyFeatureModels in enum MyFeature add the following

```
enum LoadingView {
      struct Response {
         let title: String
         let subTitle: String
      }
}
```

*In MyFeaturePresentationLogic

```
// Kata 7
func presentLoadingView(response: MyFeature.LoadingView.Response)
func hideLoadingView()
```

*In MyFeaturePresenter

```
// Kata 7
func presentLoadingView(response: MyFeature.LoadingView.Response) {
}
func hideLoadingView() {
}
```

*In MyFeaturePresentatinoLogicSpy

```
// Kata 7
var presentLoadingViewIsCalled: Bool = false
var presentLoadingViewResponse: MyFeature.LoadingView.Response?
func presentLoadingView(response: MyFeature.LoadingView.Response) {
      presentLoadingViewIsCalled = true
      presentLoadingViewResponse = response
}

var hideLoadingViewIsCalled: Bool = false
func hideLoadingView() {
      hideLoadingViewIsCalled = true
}
```

Now it will compile but it won't pass. We need to actually call the methods and for that we need to refactor fetchMovements(request:)
*In MyFeatureInteractor in fetchMovements(request:)

```
// Kata 7
      presentLoadingView()

      // Kata 4
      worker.fetchMovements(completionHandler: { (movements, error)  in
         if error == nil {
```

```
            self.presenter?.presentMovements(response:
MyFeature.Something.Response.Success(movementsResponse: movements!))
            self.hideLoadingView()
        }
    })
```

Yeah! Everything passes!!

Now to handle from MyFeaturePresenter to MyFeatureViewController

Of course start with the tests.
*In MyFeaturePresenterTests

```
// Kata 7
func testDisplayLoadingView() {
    // Given
    // When
    sut.presentLoadingView(response: MyFeature.LoadingView.Response(title: "Hi",
                                                subTitle: "Wait a moment"))
    // Then
    XCTAssertTrue(spyViewController.displayLoadingViewCalled, "showLoadingView should display Loading
View on the viewController")
    XCTAssertEqual(spyViewController.displayLoadingViewViewModel?.title, "Hi", "The loading view should
have 'Hi' for title")
    XCTAssertEqual(spyViewController.displayLoadingViewViewModel?.subTitle, "Wait A Moment", "The
loading view should have 'Wait A Moment' for subtitle")
  }

  func testHideLoadingView() {
    // Given
    // When
    sut.hideLoadingView()
    // Then
    XCTAssertTrue(spyViewController.hideLoadingViewCalled, "hideLoadingView should hide Loading View on
the viewController")
  }
```

Now it’s the usual drill.
*In MyFeatureDisplayLogic

```
// Kata 7
func displayLoadingView(viewModel: MyFeature.LoadingView.ViewModel)
func hideLoadingView()
```

The compiler will complain that you need a ViewModel for LoadingView.
*In MyFeatureModels in enum LoadingView

```
// Kata 7
struct ViewModel {
        let title: String
        let subTitle: String
    }
```

Now as usual you can let the compiler guide you through where to put the rest of the methods,
*In MyFeatureViewController

```
// Kata 7
func displayLoadingView(viewModel: MyFeature.LoadingView.ViewModel) {
}
func hideLoadingView() {
}
```

*In MyFeatureDisplayLogicSpy

```
// Kata 7
   var displayLoadingViewCalled: Bool = false
   var displayLoadingViewViewModel: MyFeature.LoadingView.ViewModel?
   func displayLoadingView(viewModel: MyFeature.LoadingView.ViewModel) {
      displayLoadingViewCalled = true
      displayLoadingViewViewModel = viewModel
   }

   var hideLoadingViewCalled: Bool = false
   func hideLoadingView() {
      hideLoadingViewCalled = true
   }
```

Now it compiles but the tests don't pass because we actually have to call the methods.
*In MyFeaturePresenter in func presentLoadingView(response:)

```
// Kata 7
let loadingViewViewModel = MyFeature.LoadingView.ViewModel(title: response.title.localizedCapitalized, subTitle: response.subTitle.localizedCapitalized)
viewController?.displayLoadingView(viewModel: loadingViewViewModel)
```

And
*In MyFeaturePresenter in func hideLoadingView()

```
// Kata 7
viewController?.hideLoadingView()
```

Congratulations! You finished Kata 7!!

Kata 8 - Entire TDD cycle for errorViews

The UX team wants to display an internetErrorView when fetchMovements() returns an internet error and a serviceErrorView when the error isn't because of the internet.
Implement the entire TDD cycle to display an InternetErrorView, hide an InternetErrorView, display a ServiceErrorView and hide a ServiceErrorView to improve functionality.

The internet error view will have a title and a subtitle.
The title will say "Oops"
The subtitle will say "You have an internet error"

The service error view will have a title and a subtitle.
The title will say "Sorry"
The subtitle will say "You have a service error"

Test the methods individually and as part of the fetchMovements() test.
You'll need to implement or refactor fetchMovements() and it's tests in some way.

There are many ways to complete this Kata and the next ones. Feel free to be creative.

// Begin the Kata. Good luck!

Kata 9 - ViewController to Router

The UX team has decided that somewhere in the ViewController there will be a button that should route to MyFeature2Scene

Implement the tests starting in ViewControllerTests and then the functionality.

Don’t worry about actually routing to the MyFeature2Scene since it’s still in the backlog.
You'll get to it in Kata 10

Kata 10 - Implement the UI

The UX team has decided:
*show the results in a tableView.
(Do they show up? If not you'll need to refactor your tests and code.
Challenge yourself to do the tests first!)

*didSelectRow should push to MyFeature2Scene.
didSelectRow should define a selectedMovement in MyFeatureIntereactor.
This object should be used by MyFeatureRouter to navigate and pass the object to the next scene.

Complete the requirement using TDD.

Congratulations!! You've finished the first 10 Katas!

Now use TDD to implement a real feature in your real project.