

w/c 30th October 2023

Introducing Persistent Data Storage

Today's lab exercise introduces you to the concept of persistent data storage - something that will be useful to you in Assignment one. Lecture Set three discusses the standard mechanisms for Persistent Data Storage.

User Defaults

****update****

In Xcode 15, playgrounds don't seem to work well with UserDefaults, so this exercise has been updated to use a full iOS project instead.

Start Xcode and create a new project. Choose an iOS App template and call the project "user defaults test".

Copy the following code into the viewDidLoad() method of your ViewController.swift file and run it.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view.

    var runCount = UserDefaults.standard.integer(forKey: "runCount")
    runCount += 1
    UserDefaults.standard.set(runCount, forKey: "runCount")
    print(runCount)
}
```

Run it, and when the app has fully started up you should see "1" printed to the console. Stop it and un the code again by clicking. This time you should see "2" printed out in the console. This value is not held in the program, but in the user defaults "database" which is stored along with the "app" built by your project. It's specific to this app, so would not be accessible from a different app if you were to create one.

Note that we used UserDefaults.standard.integer, rather than UserDefaults.standard.object - the former is easier to use because we do not have to cast the result into an Int, it already is one. There are versions of this for all the standard types. If we were to save a non-standard / custom type into UserDefaults, we'd have to use the object version, and cast it back to the correct type when we reload it

Replace the current viewDidLoad contents with the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    if let userName = UserDefaults.standard.string(forKey: "name") {
        print("Previously saved username is \"\(userName)\"")
    } else {
        print("No username previously saved")
    }
    UserDefaults.standard.set("your name", forKey: "name")
}
```

Make sure to replace the "your name" bit with your actual name in double quotes. Run this code. The first time you do so it should print "No username previously saved". Run it again. It should print the message that includes your name. Notice that when we attempt to retrieve the String from UserDefaults the result is optional (we may not have had an item in the UserDefaults database with that key), so we used if let to check the result of accessing it wasn't nil before we tried to print it out. Now that we've saved a name in UserDefaults, every time we run the code above it will retrieve the

saved name. We could of course replace that saved name with a different one, but what if we want to remove it completely from UserDefaults? How could we do that?

One way to do that is based on what we see if we look at the definition of the method, which displays the two parameters that it requires:

```
UserDefaults.standard.set(value: Any?, forKey: String)
```

Parameter one, the value that we're supplying, is an optional of any type. In other words we're allowed to supply `nil` as the parameter. Modify your code and replace the first parameter in the final line with `nil`, rather than your name in a string (don't put `nil` in double quotes). Run the code - it will again print the previously stored username. Run it a second time. What happens now?

Incidentally, another way to remove an item from the UserDefaults persistent store is to use the specific class method:

```
UserDefaults.standard.removeObject(forKey: String)
```

You can also clear the whole of the standard database (affecting only your app) with

```
UserDefaults.resetStandardUserDefaults()
```

Core Data

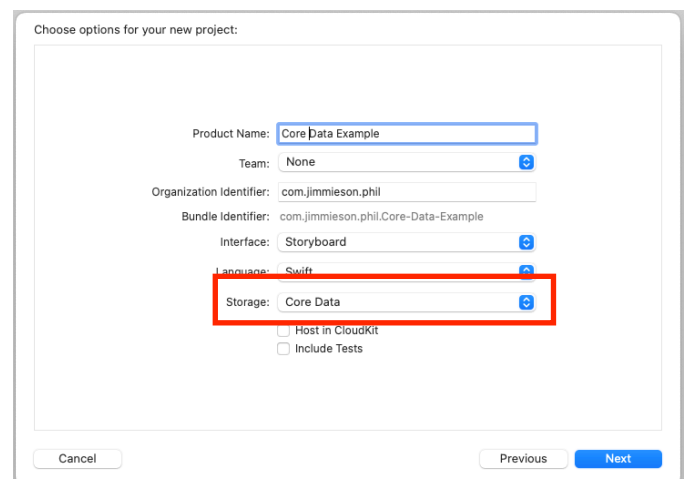
Initial Setup

This is Apple's Object-based persistent storage mechanism which is closest to that of a traditional database, in that you can store complex data. We will be creating a simple contacts app that uses Core Data to save the user information across multiple runs of the application. The app you're about to make is the first part of the second "half" of your COMP228 portfolio (actually the first app of three that forms the second half).

Start a new project. Name it "Core Data Example" and make sure you select the option for Core Data.

We will be using a table view to display our contact's information so add a table view into your view controller, dragging it out to the edges of the safe area of the view. Ctrl drag up to the yellow icon to make the table a delegate and datasource and ctrl drag to your source code to make an outlet called "myTable" (look at week 2's video for how to link a table view to your source code if you're not sure). Add a prototype cell to your table and give it the identifier "myCell".

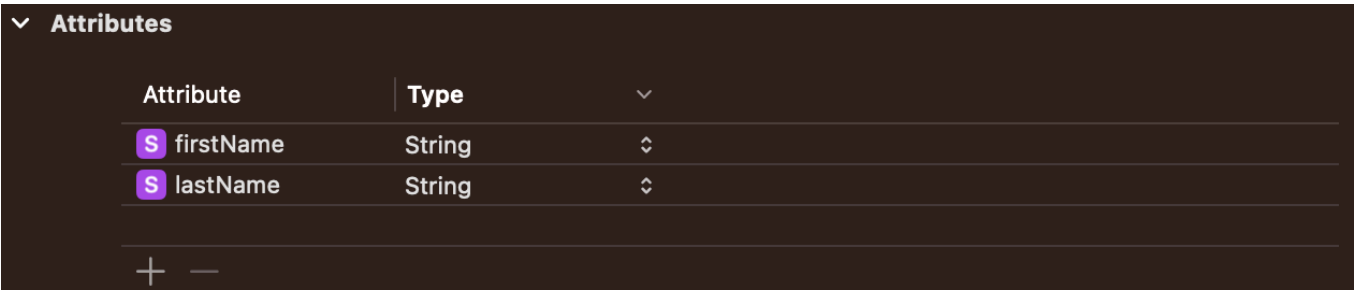
Add the protocols to your view controller (`UITableViewDataSource` and `UITableViewDelegate`) and let Xcode create the required stub methods for you to complete later.



You'll notice a new file in your project folder explorer on the left hand side that you haven't seen before. This `.xcdatamodel` file that is created is a representation of the data model, defining the types of entities, their attributes, and the relationships between them. It serves as the schema that Core Data uses to create and manage the underlying database.

When we create the file it's initially blank, so we need to create an entity that defines something we want to store, like a contact. We can add an entity with the 'add entity' button at the bottom of the page and we can click on it to rename it to something useful like "Contact". We also need to add

attributes to the entity, that define what the entity is. For a contact we may attributes like a name or someone's contact details. Click the plus button underneath the attributes section and add the attributes "firstName" and "lastName" and make them strings (don't worry if they are not in the same order as mine in the screenshot below).



That's it! Now we're ready to start using Core Data within our application! One thing to note is that when you click the "Core Data" option when creating your project, some code is automatically added into your App Delegate (see below). The code shown below (with some additional comments) should already exist in your AppDelegate.swift source code file.

```
lazy var persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "Model")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

The first function encapsulates the Core Data stack within your application, and initialises the persistent store and loads it into memory. The second function checks to see whether there has been a change to the data you are trying to save, and then saves it to persistent storage if needed.

There's nothing you need to do here, but it's good to know that if you forget to select Core Data as an option when creating a project you can add it to an existing project by manually adding this code to your existing App Delegate source file.

Contacts Application

Now that we have the initial setup completed, we can now move onto using our core data model in our contacts application! Start off by importing Core Data into your view controller:

```
import CoreData
```

In your view controller file, add the following line which allows us to have an array of our Core Data objects:

```
var contacts: [NSManagedObject] = []
```

The NSManagedObject class is a fundamental building block of Core Data in iOS, which effectively is a runtime representation of an entity that's defined in your Core Data model, which for us is "Contact".

Fetching Data

There are two main aspects of Core Data that are key to make our app function the way we intend; retrieving data from the Core Data stack and also saving data. Copy the following code and paste it into your view controller:

```
func fetchData() {
    guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
        return
    }

    let managedContext = appDelegate.persistentContainer.viewContext
    let fetchRequest = NSFetchRequest<NSManagedObject>(entityName:"Contact")

    do {
        contacts = try managedContext.fetch(fetchRequest)
    } catch let error as NSError {
        print("Could not fetch. \(error), \(error.userInfo)")
    }
}
```

Lets break this function down to see what's actually happening:

```
guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
    return
}
```

The 'guard let' statement tries to retrieve UIApplication.shared.delegate safely and stores it to a variable appDelegate if it can, but returns if an error arises.

```
let managedContext = appDelegate.persistentContainer.viewContext
```

The "persistentContainer" is an instance of NSPersistentContainer, and is responsible for managing your Core Data Stack. ".viewContext" is an instance of NSManagedObjectContext and can be thought of a workbench that allows you to use and manipulate your Core Data objects before saving them to your persistent storage.

```
let fetchRequest = NSFetchRequest<NSManagedObject>(entityName: "Contact")
```

This line describes what it is that we are fetching from our Core Data stack, which in our case is the entity called "Contact". NSFetchRequest<NSManagedObject> isn't really important to understand for our use case (yet) but it's a generic class that is used to describe a search of fetch operation that you want to perform on your Core Data class.

```
do {
    contacts = try managedContext.fetch(fetchRequest)
} catch let error as NSError {
    print("Could not fetch. \(error), \(error.userInfo)")
}
```

Finally we try to fetch the data that we setup in the lines before, and catch any errors that may come up, for example non existent entities.

Saving Data

The next aspect of Core Data we need to be aware of is how to actually save our data to persistent storage so that we can use it after the app has been closed and reopened again! Within the same view controller, paste the following function:

```
func save(firstName: String, lastName: String) {
    guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
        return
    }

    let managedContext = appDelegate.persistentContainer.viewContext
    let contact = NSEntityDescription.insertNewObject(forEntityName: "Contact", into:
managedContext)

    contact.setValue(firstName, forKeyPath: "firstName")
    contact.setValue(lastName, forKeyPath: "lastName")

    do {
        try managedContext.save()
        contacts.append(contact)
        print("SAVED")
    } catch let error as NSError {
        print("Could not save. \(error), \(error.userInfo)")
    }
}
```

You'll notice that we again access the ".viewContext" instance for the saving of the data. We initialise a new NSManagedObjectContext instance for our 'Contact' entity and insert it into our persistent storage with the following line:

```
let contact = NSEntityDescription.insertNewObject(forEntityName: "Contact", into:
managedContext)
```

NSManagedObjects have a method called ".setValue" that we can use to insert our variables that have been passed into our function into our object that we want to save. As you can see, we can use the "forKeyPath" argument to set variables for specific attributes within our Core Data Model's entity. We covered this in the recent lecture on CoreData, where I mentioned that this was the old method for working with Core Data. It does the job, but the compiler can't check that you've correctly named your attributes because they're just strings. The new method uses a Class which has properties to match the Entity's attributes (and which can be checked by the compiler). For now in this very simple case, use the old method, but be aware that a better method exists.

Once we've got ourselves a new managed object and have set values for it's attributes, we can then use the "managedContext.save()" function to save our data into persistent storage.

Linking to table view

Within our viewDidLoad function, call our "fetchData()" function so that the data is pulled from persistent storage as soon as the view controller is loaded. The user will then see it when your app starts up. Your function should look something like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    fetchData()
}
```

Next we need to link the table view to use the contacts variable we defined before. First lets define the number of rows within our table view as being the length of our contacts array.

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
    return contacts.count
}
```

To get the names of the contacts within our table view cells we can pull an item from our contacts array and then use the “.value” attribute to grab the specific information we want about a contact. To display a contact’s first and last name the function would look like this (note an additional item for the telephone number has been included which you will modify as part of the additional steps):

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: "myCell", for:
indexPath)

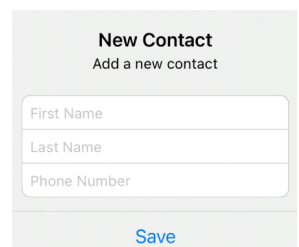
    let contact = contacts[indexPath.row]
    let firstName = contact.value(forKeyPath: "firstName") as? String
    let lastName = contact.value(forKeyPath: "lastName") as? String
    var content = UIListContentConfiguration.cell()
    content.text = "\(firstName ?? "") \(lastName ?? "")"
    content.secondaryText = "phone number"
    cell.contentConfiguration = content
    return cell
}
```

We have set the “forKeyPath” argument to be the name of the attribute that our contact entity has and returned that within our cell text (again using the old style mechanism for accessing an NSObject’s attributes).

Run the application and see if it runs. Remember you won’t see a populated table view yet as we haven’t added any data, but if there are no errors that is a good sign!

Adding a new object to Core Data

Now we’re ready to add a contact to our Core Data model (finally)! To do this lets add a button. Drag the top edge of your table view down to reduce it’s height and leave some space for a button. Drag a button from the library to just above the top right of the table view. Change it’s name from “button” to “+” and ctrl-drag to link it to our code as an action (named addContact) in the usual way. We are going to be using a UIAlertController to allow us to add some input to our application. A UIAlertController is a popup that you will often see when you have to agree to terms etc, but we can use them to add information without cluttering up our application with input fields that we don’t need very often.



```
let alert = UIAlertController(title: "New Contact", message: "Add a new contact",
preferredStyle: .alert)
```

Add the above line into your button function, which creates a new alert controller programmatically within our code. We can then use the “.addTextField” attribute of our alert controller to control how many input fields we want to appear in our alert.

```
alert.addTextField { textField in
    textField.placeholder = "First Name"
}
alert.addTextField { textField in
    textField.placeholder = "Last Name"
}
```

Now that we have the text fields for our alert controller, we can start to think about adding a button to the alert controller. Place the following button code within the same addContact button function as the code for the alert controller.

```
let saveAction = UIAlertAction(title: "Save", style: .default) { [unowned self]
action in
}
```

We don't need to worry about what the code is doing in depth, but it is good to know that the "unowned self" closure is being used here to tell Swift that "self" should not be strongly captured within the closure, preventing a retain cycle memory leak.

Next we can grab text from the text fields just like we can in any other circumstance. We will be using a guard statement to make sure we can safely unwrap any values that may be within the alert's text fields (note the commas between multiple clauses). After grabbing the textfield objects from the alert's array of textfields, we access the text property of each to get at the strings. We then use our save function we created earlier to put our new information into persistent storage. The full button code is below (but **don't copy and paste this all into your app - take everything but the first line and use the original first line from your button method so as to leave the storyboard link in place.** You'll need to leave the first line of your original action function alone, otherwise you'll break the link from the storyboard):

```
@IBAction func addContact(_ sender: Any) {
    let alert = UIAlertController(title: "New Contact", message: "Add a new
contact", preferredStyle: .alert)

    alert.addTextField { textField in
        textField.placeholder = "First Name"
    }
    alert.addTextField { textField in
        textField.placeholder = "Last Name"
    }

    let saveAction = UIAlertAction(title: "Save", style: .default) { [unowned self]
action in
        guard let firstNameField = alert.textFields?[0],
            let lastNameField = alert.textFields?[1],
            let firstName = firstNameField.text,
            let lastName = lastNameField.text else {
            return
        }

        self.save(firstName: firstName, lastName: lastName)
        self.myTable.reloadData()
    }

    alert.addAction(saveAction)
    present(alert, animated: true)
}
```

Note that we call the "present" method at the end of our button action function, as without this the alert controller will never be shown when our button is pressed.

After all of that, run your app and add a couple of names to your contacts list. Stop your app and run it again. Those names you just saved should appear immediately - retrieved from persistent storage where they were saved between runs of your app!!

Your contribution (for an extra mark)

Swipe to Delete

For the first step for the additional mark for this App, you need to add a “swipe to delete” function to our table view, that also deletes the selected item from persistent storage. Below is a function that allows you to delete a specific row from both the table view and core data. Your job is to implement the “swipe to delete” feature:

```
func delete(indexPath: IndexPath) {
    // Delete the object from Core Data
    guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
        return
    }

    let managedContext = appDelegate.persistentContainer.viewContext
    let contactToDelete = contacts[indexPath.row]
    managedContext.delete(contactToDelete)

    do {
        try managedContext.save()
        // Remove the object from the array
        contacts.remove(at: indexPath.row)
        // Remove the table view row
        myTable.deleteRows(at: [indexPath], with: .fade)
    } catch let error as NSError {
        print("Could not delete. \(error), \(error.userInfo)")
    }
}
```

Hint: We have already used a table view delete method in one of the earlier practical sessions. Google and demonstrators are also available if you’re not sure!

Telephone number

The 2nd step for the additional mark is for you to add support for a contact’s telephone number to the App. You’ll need to update the Entity and all the places in the code that currently work with the first and last name to add support for the additional attribute.

When you’ve finished, compress your project and save it for the second part of your portfolio.

Basic Program full source code (for reference only – don't copy and paste this as it will break your table outlet and addContact button action method).

```
import UIKit
import CoreData

class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {

    var contacts: [NSManagedObject] = []

    @IBOutlet weak var myTable: UITableView!

    //MARK: - Core Data methods

    func fetchData() {
        guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
            return
        }

        let managedContext = appDelegate.persistentContainer.viewContext
        let fetchRequest = NSFetchRequest<NSManagedObject>(entityName: "Contact")

        do {
            contacts = try managedContext.fetch(fetchRequest)
        } catch let error as NSError {
            print("Could not fetch. \(error), \(error.userInfo)")
        }
    }

    func save(firstName: String, lastName: String) {
        guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
            return
        }

        let managedContext = appDelegate.persistentContainer.viewContext
        let contact = NSEntityDescription.insertNewObject(forEntityName: "Contact", into: managedContext)

        contact.setValue(firstName, forKeyPath: "firstName")
        contact.setValue(lastName, forKeyPath: "lastName")

        do {
            try managedContext.save()
            contacts.append(contact)
            print("SAVED")
        } catch let error as NSError {
            print("Could not save. \(error), \(error.userInfo)")
        }
    }

    //MARK: - table view methods

    func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
        return contacts.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "myCell", for: indexPath)

        let contact = contacts[indexPath.row]
        let firstName = contact.value(forKeyPath: "firstName") as? String
        let lastName = contact.value(forKeyPath: "lastName") as? String
        var content = UIListContentConfiguration.cell()
        content.text = "\(firstName ?? "") \(lastName ?? "")"
        content.secondaryText = "phone number"
        cell.contentConfiguration = content
        return cell
    }

    @IBAction func addContact(_ sender: Any) {
        let alert = UIAlertController(title: "New Contact", message: "Add a new contact", preferredStyle: .alert)

        alert.addTextField { textField in
            textField.placeholder = "First Name"
        }
        alert.addTextField { textField in
            textField.placeholder = "Last Name"
        }

        let saveAction = UIAlertAction(title: "Save", style: .default) { [unowned self] action in
            guard let firstNameField = alert.textFields?[0],
                  let lastNameField = alert.textFields?[1],
                  let firstName = firstNameField.text,
                  let lastName = lastNameField.text else {
                return
            }

            self.save(firstName: firstName, lastName: lastName)
            self.myTable.reloadData()
        }

        alert.addAction(saveAction)
        present(alert, animated: true)
    }
}
```

```
//MARK: - View Controller methods

override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view.
    fetchData()
}
}
```

v1.2