Jeffrey Bergier

Traveler @ 🌏
iOS Developer @ Topology Eyewear

Past Life:
UX Designer @ Riverbed (4 years)
iOS Dev Instructor @ General Assembly

@jeffburg

jeffburg.com

www.saturdayapps.com

WaterMe

Plant Watering Reminders

Gratuity

The Simple Tip Calculator

Carrier 8:45 AM 100%

WaterMe Plants

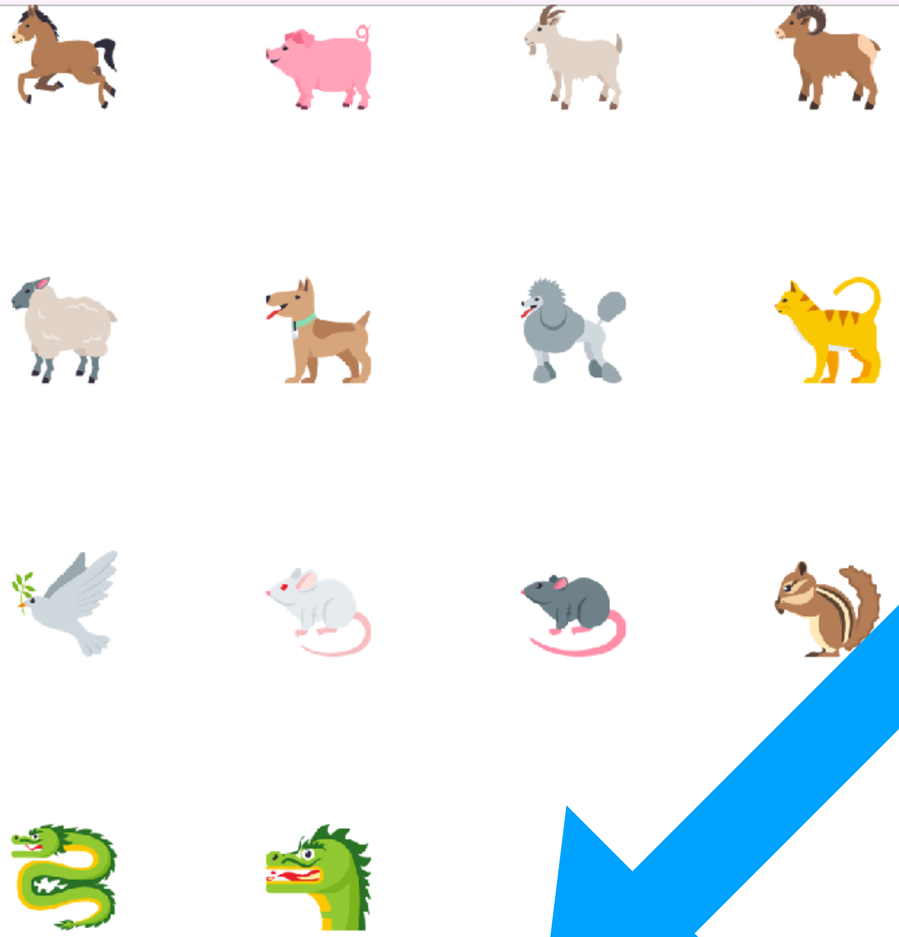
Late

Avocados
Water Plant

Pears
Water Plant

Lemon Tree
Water Plant

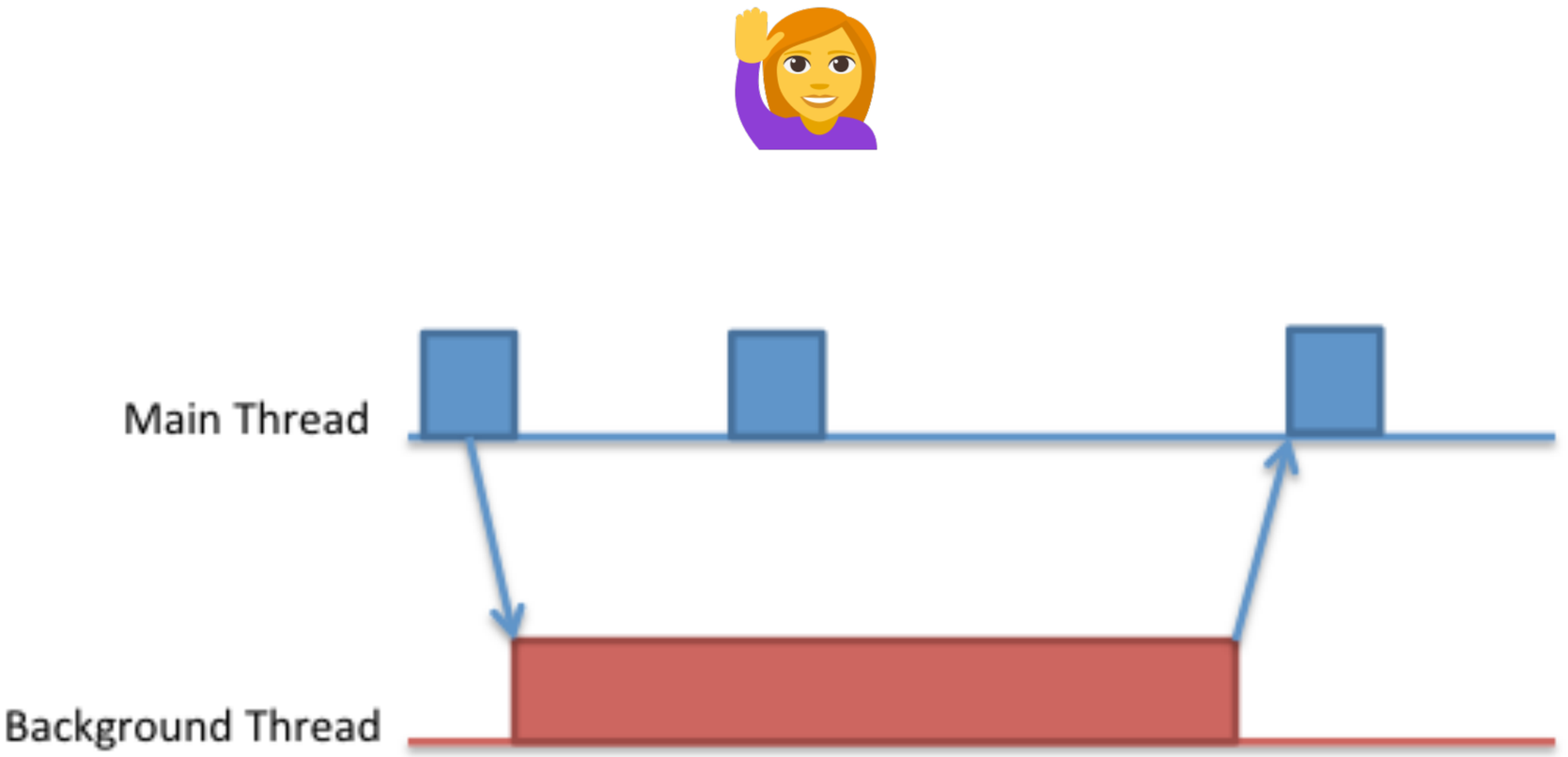
Choose an Emoji Cancel



Emoji Provided by EmojiOne

What are common asynchronous operations?

Why do we use asynchronous code?


Main Thread
Background Thread

But thats very iOS specific.

I like to think that asynchronous code is about perspective.

Our Perspective

```
let url = URL(string: "https://www.mydomain.com/myendpoint")!
let task = URLSession.shared.dataTask(with: url)
{ data, response, error in
    // process data
}
task.resume()
```

Their Perspective

```
let url = URL(string: "https://www.mydomain.com/myendpoint")!
let task = URLSession.shared.dataTask(with: url)
{ data, response, error in
    // process data
}
task.resume()
```

Thread 1

Thread 2

Thread 3

The important thing about this syntax is its close to a function.

Data in / Data out

```
let url = URL(string: "https://www.mydomain.com/myendpoint")!
let task = URLSession.shared.dataTask(with: url)
{ data, response, error in
    // process data
}
task.resume()
```

Data Out

Data In

But it wasn’t always this great. Lets go back in time.

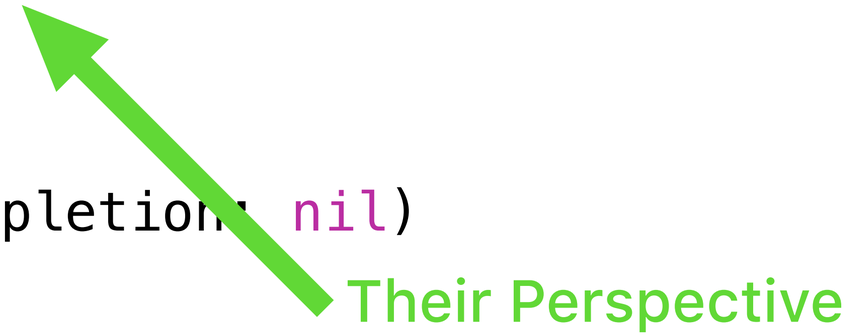
```
#pragma mark NSURLConnection Delegate Methods

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response {
    // A response has been received, this is where we initialize the instance var you created
    // so that we can append data to it in the didReceiveData method
    // Furthermore, this method is called each time there is a redirect so reinitializing it
    // also serves to clear it
    _responseData = [[NSMutableData alloc] init];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
    // Append the new data to the instance variable you declared
    [_responseData appendData:data];
}

- (NSCachedURLResponse *)connection:(NSURLConnection *)connection
                  willCacheResponse:(NSCachedURLResponse*)cachedResponse {
    // Return nil to indicate not necessary to store a cached response for this connection
    return nil;
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    // The request is complete and data has been received
    // You can parse the stuff in your instance variable now

}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error {
    // The request has failed for some reason!
    // Check the error var
}
```

In my opinion, we still deal with View Controllers in this old-fashioned way.

```swift
class ContactListViewController2: UIViewController {

    var data: [Contact] = []
    var editedIndexPath: IndexPath?
    @IBOutlet weak var tableView: UITableView!

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        // note this doesn't work on iPad because the presentation is a form sheet.
        if let editedIndexPath = self.editedIndexPath {
            // a row was edited, so we need to reload it in an animated way
            self.tableView.reloadRows(at: [editedIndexPath], with: .automatic)
        } else if let selectedIndexPath = self.tableView.indexPathForSelectedRow {
            // if nothing was edited, then we just need to deselect the selected row in an animated way
            self.tableView.deselectRow(at: selectedIndexPath, animated: true)
        }
        self.editedIndexPath = nil
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if
            segue.identifier == "ContactCellSelectedSegue",
            let cell = sender as? UITableViewCell,
            let indexPath = self.tableView.indexPath(for: cell),
            let destVC = segue.destination as? ContactEditViewController
        {
            // prepare the contact edit vc for presentation
            let contact = self.data[indexPath.row]
            destVC.contact = contact
        }
    }

    @IBAction func unwindFromContactEditViewController(_ segue: UIStoryboardSegue) {
        if
            segue.identifier == "ContactEditUnwindSegue",
            let selectedIndexPath = self.tableView.indexPathForSelectedRow,
            let contactEditVC = segue.source as? ContactEditViewController
        {
            let editedContact: Contact! = contactEditVC.contact
            let originalContact = self.data[selectedIndexPath.row]

            // check to see if the new contact is different than the old one
            if editedContact != originalContact {
                self.data.remove(at: selectedIndexPath.row)
                self.data.insert(editedContact, at: selectedIndexPath.row)
                self.editedIndexPath = selectedIndexPath
            }
        }
    }
}
```

What if we “upgraded” this syntax in a similar way to networking code?

```
let contact = self.data[indexPath.row]
let contactEditVC = ContactEditViewController.newVC(contact: contact)
{ contactEditVC, editedContact in
    // process the data
}
self.present(contactEditVC, animated: true, completion: nil)
```



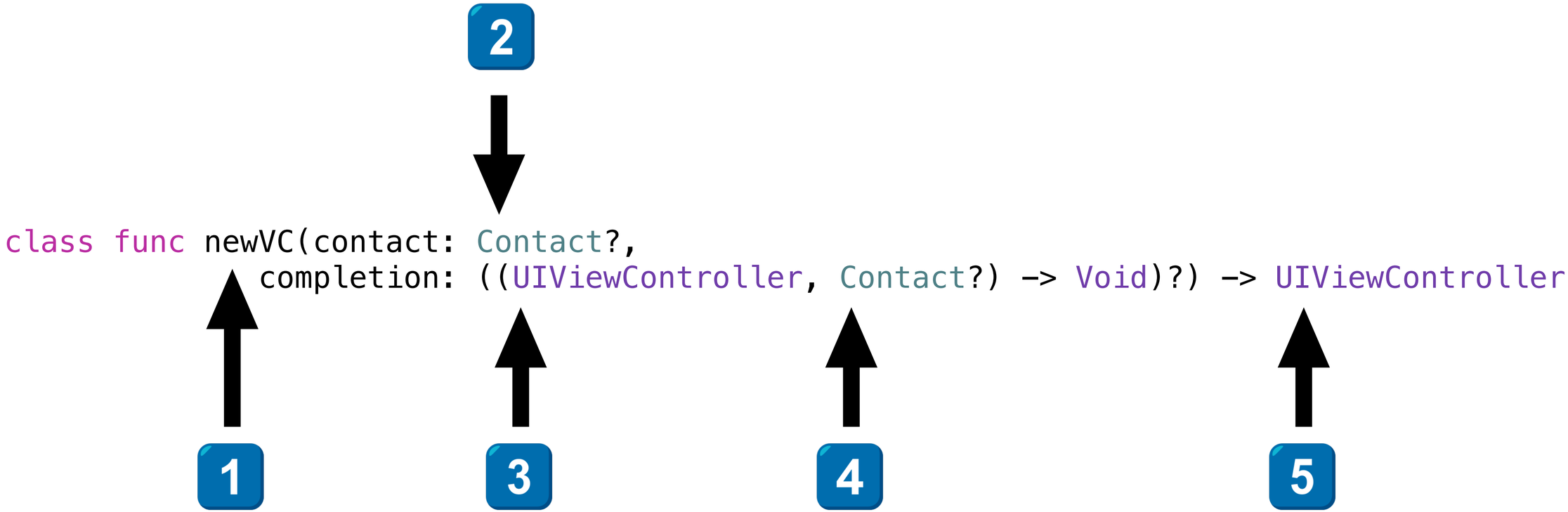
```
let contact = self.data[indexPath.row]
let contactEditVC = ContactEditViewController.newVC(contact: contact)
{ contactEditVC, editedContact in
    // process the data
}
self.present(contactEditVC, animated: true, completion: nil)
```

Data In

Data Out

In this context, UIViewController presentation is the most asynchronous operation you will perform.

Method Sinature



Method Signature

1. Named ‘newVC’ to not conflict with ‘new’ method on NSObject.
2. Pass in anything the view controller needs to do its job.
    1. Simple data, complex database manager, flow controller, etc.
3. For Swift reasons, we need to pass a UIViewController into the completion handler so we can dismiss it.
4. Pass out any data you want.
    1. Success Boolean, Simple data type, result type, enum, etc.
5. Return plain old UIViewController
    1. This prevents the presenting view controller from messing with it.
    2. Also, allows you to use container view controllers:
        1. UINavigationController, UITabBarController
        2. Your custom ones

And Lets Look at the Data Processing Code.

```
class ContactListViewController: UIViewController {

    var data: [Contact] = []
    @IBOutlet weak var tableView: UITableView!

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let contact = self.data[indexPath.row]
        let vc = ContactEditViewController.newVC(contact: contact) {
            [unowned self] vc, contact in
            guard let contact = contact else {
                // if we don't have a new contact, we just need to deselect the selected row
                vc.dismiss(animated: true) {
                    tableView.deselectRow(at: indexPath, animated: true)
                }
                return
            }
            // if we do, we need to update our data source
            self.data.remove(at: indexPath.row)
            self.data.insert(contact, at: indexPath.row)
            // and also reload the appropriate row
            vc.dismiss(animated: true) {
                tableView.reloadRows(at: [indexPath], with: .automatic)
            }
        }
        self.present(vc, animated: true, completion: nil)
    }
}
```

This Approach Has a Lot of Advantages

- Presentation and dismissal no longer just “happen” to the presenting view controller.
    - Rather, the presenting view controller is in full control.
    - No more hacks in viewDidAppear because we are in full control of when we appear again.
- All code is in a single context.
    - No need to keep finding the Selected IndexPath, we just capture it in a our closure.

More Abstract Advantages

- The presented view controller can do anything it wants in the view controller presentation chain.
  - You can use flow controllers
  - UINavigationController with multiple steps
- The View Controller is also easily replaceable. As long as it follows the same API contact, it works the same.
  - Useful for A/B Testing.
  - Putting your view controllers in frameworks to control their complexity.

There are Some Rules Though

- Only works for Modal Presentation
  - self.present & self.dismiss
  - NOT self.navigationController.push
- Completion Handler
  - Must be called Once and only Once by the presented view controller
  - Just like any completionHandler based programming, the implementation logic can get complicated and it can be hard to make sure this rule is followed.
- Be careful of retain cycles

So, what does the ContactEditViewController code look like?

```
class ContactEditViewController: UIViewController {

    class func newVC(contact: Contact?,
                     completion: ((UIViewController, Contact?) -> Void)?) -> UIViewController
    {
        // create your vc however you want. Storyboards work
        let vc = ContactEditViewController()
        // configure your vc
        vc.contact = contact ?? Contact()
        vc.completionHandler = completion
        return vc
    }

    private var contact: Contact?
    private var completionHandler: ((UIViewController, Contact) -> Void)?

    @IBAction func saveTapped(_ sender: Any) {
        self.completionHandler?(self, self.contact)
    }

    @IBAction func cancelTapped(_ sender: Any) {
        self.completionHandler?(self, nil)
    }
}
```

Demo

But, there's an advanced mode. Its even better!

Have You Used This?

```
public protocol UIViewControllerTransitionCoordinator : UIViewControllerTransitionCoordinatorContext {
    public func animate(alongsideTransition animation: ((UIViewControllerTransitionCoordinatorContext) -> Swift.Void)?,
                                            completion: ((UIViewControllerTransitionCoordinatorContext) -> Swift.Void)) -> Bool
}
```

UIViewController has a property for its UIViewControllerTransitionCoordinator

But the property can be NIL so you need to be careful with it.

```
extension UIViewController {

    typealias ContextClosure = (UIViewControllerTransitionCoordinatorContext?) -> Void

    func animateAlongSideTransitionIfPossible(_ animate: @escaping ContextClosure) {
        self.animateAlongSideTransitionIfPossible(animate, completion: nil)
    }

    func animateAlongSideTransitionIfPossible(_ animate: @escaping ContextClosure,
                                              completion: ContextClosure?)
    {
        if let tc = self.transitionCoordinator {
            tc.animate(alongsideTransition: animate, completion: completion)
        } else {
            animate(nil)
            completion?(nil)
        }
    }
}
```

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let contact = self.data[indexPath.row]
    let vc = ContactEditViewController.newVC(contact: contact) {
        [unowned self] vc, contact in
        guard let contact = contact else {
            // if we don't have a new contact, we just need to deselect the selected row
            vc.dismiss(animated: true)
            self.animateAlongSideTransitionIfPossible() { _ in
                tableView.deselectRow(at: indexPath, animated: true)
            }
            return
        }
        // if we do, we need to update our data source
        self.data.remove(at: indexPath.row)
        self.data.insert(contact, at: indexPath.row)
        // and also reload the appropriate row
        vc.dismiss(animated: true)
        self.animateAlongSideTransitionIfPossible() { _ in
            tableView.reloadRows(at: [indexPath], with: .automatic)
        }
    }
    self.present(vc, animated: true, completion: nil)
}
```

Demo

Amazing, Right?

Thank You

Q&A

@jeffburg

jeffburg.com

https://github.com/jeffreybergier/MarchTokyo2018