iPhreaks

092 iPS Unit Testing with NatashaTheRobot

[This episode is sponsored by Hired.com. Every week on Hired, they run an auction where over a thousand tech companies in San Francisco, New York and L.A. bid on iOS developers, providing them with salary and equity upfront. The average iOS developer gets an average of 5-15 introductory offers and an average salary offer of $130,000/year. Users can either accept an offer and go right into interviewing with a company or deny them without any continuing obligations. It's totally free for users, and when you're hired they also give you a $2,000 signing bonus as a thank you for using them. But if you use the iPhreaks link, you'll get a $4,000 bonus onset. Finally, if you're not looking for a job but know someone who is, you can refer them on Hired and get a $1,337 bonus as thanks after the job. Go sign up at Hired.com/iphreaks]

[This episode of iPhreaks is brought to you, in part, by Postcards. Postcards is the simplest way to allow you to feedback from right inside your application. With just a simple gesture, anyone testing your app can send you a Postcard containing a screenshot of the app and some notes. It's a great way to handle bug reports and feature requests from your clients. It takes 5 minutes to set up, and the first five postcards each month are free. Get started today by visiting www.postcard.es]

CHUCK:

Hey everybody! Welcome to episode 92 you have the iPhreaks show. This week on our panel, we have Alondo Brewington,

ALONDO:

Hello from Carolina.

CHUCK:

Jaim Zuber.

JAIM:

I think I'm secure enough in my adulthood to wear a Care Bear band-aid.

CHUCK:

Nice! I'm Charles Max Wood from DevChat.TV. And this week, we have a special guest, Natasha – I should have asked you how to say your last name.

NATASHA:

You can just call me Natasha the Robot.

CHUCK:

Awesome. Do you want to introduce yourself really quickly?

NATASHA:

Sure! So, I'm an iOS developer. I currently work at Capital One. And I usually just learn a lot and I write about what I'm learning on my blog, Natasha the Robot. Recently since Swift came out, I've been really focusing on it. So lot of Swift stuff – I have a Swift newsletter. That's pretty much it. Currently learning WatchKit as well – a little bit.

ALONDO:

So, Natasha, are you actually doing Swift projects in – at Capital One or is this something you're

just exploring as a way to get up to speed?

NATASHA:

Yeah, we switched to Swift pretty fast – pretty much as soon as it came out. So I've been working on it pretty much full time and on the side since it came out.

JAIM:

Yeah, I really liked the blog. So I started doing Swift development full-time about two months ago and about half the questions I end up going to Google for ended up on your blog. So, thank you for that.

NATASHA:

Oh! Awesome! [Chuckling] Usually if I Google something and it doesn't exist, I try to write a blog post.

CHUCK:

Yeah! Natasha the robot knows everything.

ALONDO:

Well is that a different approach than - to you blog than prior to Swift or is this something you've always done since you've had the blog?

NATASHA:

Yes, it's something I've always done. So there's stuff about like Ruby on Rails development from about two years ago.

CHUCK:

Yey! Rails!

NATASHA:

It's just so way for me to solidify my knowledge, so I often feel if I learn really fast, then I feel like I lose it really quick. Most of it is actually a reference for myself. You know that feeling when you solve something and you're good but then a few months later – even few weeks later, you have the same problem and you're like, "I used to know how to do this."

So, a lot of times I just google my own blog. For Swift, like unwrapping multiple optionals – I google that one all the time.

CHUCK:

Yeah, I have to admit that I used the same tactic where I run into an error or something. So I'll throw a quick blog post up there and then I Google my own website because I know the answer is there.

NATASHA:

Yup.

CHUCK:

So, you have a post on here. I have to admit, I love the idea of testing code and this is one area that – as I've been learning Swift I've just been ignoring mostly because the tutorials that I've worked through don't really address it. So – and that's unit testing or just testing in general with Swift.

testing:

a quick look at quick.” And I'm wondering, how did you find that – how did you get started with it?

NATASHA:

So I actually learned about it through people like Pivotal labs. So, one of the – I think the main contributor is someone from Pivotal and a person from Facebook. So, I heard about it. I was just starting a brand new project in Swift. I wanted to have better frameworks for testing, so quick was the only obvious choice. It's a lot more like RSpec, which I like so we just went with it.

ALONDO:

So Quick is a framework for testing?

NATASHA:

Yeah! So I don't know if you're familiar with Ruby on Rails RSpec – kind of BDD style testing where it's more readable than XC tests. They basically say you have a context describe statement then it’s like the different asserts are like it statements.

CHUCK:

Yeah! I really like the style. JavaScript also has Jasmine, which is kind of like that.

NATASHA:

Yeah, exactly!

CHUCK:

I'm trying to think of some of the other frameworks similar to it that other languages have but those are the ones that I think people are going to be most familiar with in the web space is RSpec and then there's–

NATASHA:

Kiwi for iOS.

CHUCK:

Oh, yes!

ALONDO:

I was trying to remember. So it’s pretty similar to this Kiwi app? I’ve actually used Kiwi a little bit but I haven't really taken a full dive into using unit testing–

NATASHA:

Yeah! Kiwi was a little bit awkward because it used the blocks syntax with Objective-C. So it's weird to right. But Quick is – I mean Swift is so much easier to work with.

JAIM:

Okay, so Quick is Swift native?

NATASHA:

Yeah.

JAIM:

Or just has better bindings.

NATASHA:

I think it's – Yeah, it's Swift. I know they have some Objective-C code to compare it to different types like classes but I think mostly it's in Swift. And it's made for Swift, specifically, so it's a lot more – it's created for the trailing closure syntax. It's a lot more Swifty.

JAIM:

Very cool! Maybe we should step back a little bit because even if I love writing unit test with just your basic XCT test, that's forward thinking in the iOS world. Not as much as it was a year or two ago maybe, but can you explain BDD (Behaviordriven Development)? What advantage it gives you over civil unit tests?

NATASHA:

Yeah, so it just – makes to keep the user in mind. So when you're writing code, you're thinking about "Why am I writing this code? What is the purpose of it?" And then, you end up instead of just writing like a straight up unit test, you end up thinking about how to architect – it helps you architecture your code in a better way because you're thinking more about like what behavior is the user going to take.

CHUCK:

The way that I explain it to people is "Outside In". So you start at the level, like Natasha said, where the user's actually hitting the application and then you work your way down. Where most testing frameworks push you the other way where you write tests around your data models or your entities that represent movement or other responsibilities in the system and then work your way back out. And so, by working from the outside in, you have a user-centric focus instead of a code-centric focus.

JAIM:

Okay. So like code-centric – so I create a view controller, I do the setup in my class, I check properties on it. With BDD, I'm doing something little more user-focused. The thing that was hard for me to wrap my brain around when I tried it, and I haven't done a whole lot with BDD iOS, was how do you interpret the user type commands. What would be a typical user command that you would test if you're building a view controller.

NATASHA:

Ah, so just to clarify I think Quick is actually more on the unit testing level but it's more readable. For iOS, one of the frameworks I like from more – the integration tests more user-centric is KIF. And with KIF, you basically say like, user taps on the text box, enters this thing, they click Save button, it should go to the next screen. So you could have really high-level tests with that.

ALONDO:

Now in comparison to Kiwi, one of the things I like about it when I was doing a bit of work with the ability to nest tests – so around a particular scenario. Is that available in Quick as well?

NATASHA:

Yeah definitely! So, you have the describe statement where you can say the method name, then you can have a different context. So, if you're testing your API you're like, "API was successful. API had an error." And then within the "API had an error" context you can have it statements around maybe every single error if they're handled differently.

CHUCK:

Very nice. So you said unit test, are we talking view controllers then or are we talking data access layers and API layers?

NATASHA:

You should test as much as possible. So, everything! [chuckling] Yeah, there is a way to test view controllers but models are probably the easiest to tests. And then view controllers in iOS are more difficult to test because you have to do some dependency injection. But, if you have clear models or view models and break out your code, that's a lot easier to test.

CHUCK:

That makes sense. What do you use for the Mocking or dependency injection anyway? What do you inject in in order to you avoid the issues that you're going to have by going full stack?

NATASHA:

Yeah, usually, if you have a view controller then you probably have to dependency and check like your API service model so you can kind if fake what happens. So you can fake an error state or a success state.

JAIM:

Now, in Objective-C, the default for creating mocks to things like that would be OC mock. Is there anything coming down the pipe in Swift that's similar?

NATASHA:

Yeah, I'm sure there is. I haven't really used anything so I haven't really needed a full-on mocking network.

JAIM:

Okay, so how do you create the data like if you want to create an error case from API and test that your view controller's behaving incorrectly, what would you do?

NATASHA:

Yeah, so that would be more of the dependency injection. So you make it public your – whatever your API server layer, you can make it public and then you inject it, you can sub-class it and injected to be that one and then you can fake their response in your tests. You have a lot more control over it by sub-classing it in your tests.

JAIM:

Okay, so you take your API, you create a sub-class, you override one of the methods and you can–

NATASHA:

Yeah exactly!

JAIM:

You can verify something happened to it.

NATASHA:

Yeah! [Crosstalk 11:22] Your tests are really easy because you're just like, "Oh it was true!" – like this was called. You don't have to test that are actually went to the API or any complicated logic. And you don't want to test that in your view controller. You just want the quick "If this happened, what's the reaction?"

JAIM:

Okay, that's like old school TDD thing from the old Java guys, what they were doing. And I didn't see a lot of it with Objective-C because it was so bulky to create a sub-class. I think with Swift it's so easy to create a class and just do a method, and it was something that I see people doing a lot more. I don't see a lot of the mocking frameworks or much of a need for one. I haven't really used one; I do the same thing you talked about. I'll sub-class it, create a method and just verify that it was called.

NATASHA:

Yeah! And then with Swift, it's a little bit more functional programming or it makes you do that more so you can – you end up writing code that's also more independent.

CHUCK:

What do you mean more independent?

NATASHA:

Just like – yeah, I think this is not just functional programming but just like separation of concerns where you inject dependencies into the method versus having some kind of global constant that you have to manipulate [inaudible 12:38] that relies on it. So, I try to make my methods like self – you have an input, you have an output, hopefully no side effects like a global state type of stuff.

JAIM:

Yeah, that's definitely one of the key features of writing tests for code is having simple inputs and outputs – something that you can test.

NATASHA:

Yeah exactly.

JAIM:

Have a method and you're incrementing some global counter. That's a nightmare to test and it's just bad practice in general, but having to write a test for it actually really flushes that type of poor design out pretty quickly.

ALONDO:

So let's talk a little bit more about Quick spec. Just trying to understand how you create your tests because I'm used to creating a system under test directly. With Quick spec, you have like context, you have describes. Can you go a little bit and tell us what described is? The RSpec.

NATASHA:

Yeah so usually it's use just to have the name of the method or if it's a instance method, you put a period in front of the method, but it's very – it makes it very easy to scan your code then be like, "Okay, I'm describing this method," and then you can go to your code that you know exactly what it corresponds to. And then for – if you're using XCTests, like sometimes I would put underscores to separate them out. But then within each described block, you have different contexts. And a context is, again, a more high-level summary. So you have your high-level function so that maybe it could be like fetch my object, fetch user. So you would have described that fetch user. So now, you know exactly that it corresponds to the fetch user method and your API service. And then, your context would be "The fetch was successful, fetch was unsuccessful." And within each context, you would have it statements and these are the assertions of what you actually expect.

So when fetch is successful, if you're testing the view controller, you would say it goes to the next

page. And then you would do that testing if it – if the context says that there's an error, I – you would say it shows an error. And then, sometimes there could be – within the context, you can actually have other contexts even. So it's pretty incredible and nested but you can be very clear on what errors – maybe if it's – if different errors are handled differently you can have those different contexts within that context.

ALONDO:

Okay, so if I'm hearing you correctly, describe is your top-level thing. So you do something like describe my view controller or maybe something a little more clear English. Your context is different things like when my view happens and the network fails, then below that we do the “it” where we actually do that assertions. And if we were using XCTest, we do an XC assert not nil or it’s true.

NATASHA:

Exactly.

ALONDO:

Okay. So we can Nest different context within describe and–

NATASHA:

And within other contexts.

ALONDO:

Okay so we have context within context.

NATASHA:

Yep!

ALONDO:

And ideally, what shows out in the log is similar to a sense. So it’s readable.

NATASHA:

Exactly! Yeah, it reads like a sentence.

CHUCK:

In some cases, you actually wind up where you're not just asserting the direct return value or whatever but you're also asserting that the right side effects occurred. And so, you can have multiple its that one says, “I got this return value” and the other one says, “That one side effect is…” – the error is a good example that but it updated this other thing it went and updated core data or whatever.

NATASHA:

Yeah, pretty much anything in your code that has side effects in returned statements you want to have it statements for it.

JAIM:

Now inside of your “it” statements, can’t you use the XCT asserts? Or do you use this on assertion framework.

NATASHA:

Yeah so QuickSpecs – they actually sub-class from XCTest so everything is available but it's

actually it better to use their own assertions because they're just more readable. So their assertions are things like expect my result to be five. So it's a lot more readable and it's more of a sentence than like "assert nil" or "assert equal". You have the clear this is what I expect.

ALONDO:

In that regard, I've just – I'm just reading through it and I see that "Be Truthy" expectation. I don't know is that true? Or is that sort of true? It's just reading funny.

NATASHA:

I usually just be true. I have a new style on, but one thing you can see in there is that they do have things to eventually be truthy is so you can have different matchers. If you have an API request to have a synchronous – it'll keep checking until it gets the result. That's really nice. I know XCTests has added the synchronous testing, but if you tried it's clunky it's not as clean as like to eventually be true it's – you have to say "start" and then this is what I expected then "end".

CHUCK:

So one thing that I'm curious about is that there are certain parts. I work mostly on the web and then I do some mobile stuff. And I'm just, like I said, getting into this arena where want to be testing my iOS apps. Are there parts of your iOS app that you don't test either because they don't fit this paradigm well or because they're too brittle to write the test against?

NATASHA:

Yeah, if it's too brittle, you probably should rewrite it. Rewrite the code. I have more – I work more on a newer project where we started with testing so it hasn't been an issue. Probably it just – if you have other dependencies, sometimes if you have different – like CocoaPod, then they don't support dependency injection then it makes it really difficult to test.

CHUCK:

So do you just test the boundaries of those then and then count them to having written good code?

NATASHA:

If you do integration tests, that should take care of a lot of that. Like KIF or Cucumber tests more of the higher-level functionality and it should test all those cases, hopefully.

CHUCK:

Yeah, the question was trying to ask I guess is that, in so in the case of Rails, the views, which would be the user interface or the UI view stuff, it can vary sometimes depending on what you're doing with it as to what shows up and how you can test that. Do you put marks behind it and test it or do you just skip it?

NATASHA:

Yes, so I actually like to use view models. So with view models you can put any logic that goes into the view and not a separate model like class and that's very easy to test because you're like, "Oh, the user name is blank," then it should be read there like yeah validation code.

But, actual visual testing, like if something is off in design, or if something is off by five pixels, that's more human testable so we have QA people who can take a look and find all the cases.

Yeah, so one thing is your code is only as – actually, I don't think it's possible to test everything because we're human so we're not going to think of every – each case. So one of the rules I had in my old job when I Did Ruby on Rails is if you find a bug, add a test for it so there's going to be bugs even if you have perfect tests or you think you do, there's always going to some random scenario

that you didn't think about. So, in that case, when you encounter the bug, just add a test and to prevent it from happening again.

CHUCK:

Yeah that makes a lot of sense.

NATASHA:

And if you use KIF or Cucumber, that does a lot more of the work on the view layer because it will actually go in and put input into a text field to make sure there is a text field and click on the button and make sure it goes to the next screen. So, that type of testing helps with the view stuff as well.

JAIM:

So, along the lines of unit testing in Swift, one of the first questions I had was testing this class we created. How do we represent the class in our test? Do we use just a standard class? Do we make it an optional? Or do we make it an implicitly unwrapped optional with the exclamation points? One of the blog posts I came into was one of your talks – one of your blog posts about that. Can you tell us your approach for that?

NATASHA:

Yeah so a lot of – because there's no in that inert method in tests. If you have some kind of set up method, you would declare your global stuff that you're testing and as explicitly unwrapped optionals if the things you want to test are optionals. And then, in the setup method or before each method, you would reassign it every time.

JAIM:

Okay, so you're declaring a var – would it still be a var in quick spec?

NATASHA:

Yeah, it has to be a var because you're reassigning.

JAIM:

Okay, each time through. So the class is created once but each time you go through it calling a setup, which sets the value.

NATASHA:

Yeah!

JAIM:

So, Okay.

NATASHA:

It reassigns it.

JAIM:

So you got the exclamation point with that. Do you usually avoid like the plague? [Crosstalk 22:23]

NATASHA:

Yeah, do not put it in real code.

JAIM:

Okay, so exclamation good. Test code, bad real code, okay. We are on the same page there. That's good. But since setup happens before anything else–

NATASHA:

And the nice thing about Quick is they will actually compare optionals for you. So if you do XCTests it's annoying because if you have a string, that's an optional result and it equals to non-optional string. You have to double check. You have to do extra work where you have to force unwrap. With Quick they do the work for you so you can compare an optional string to non-optional one.

JAIM:

Oh, that's very cool. Because what happens if you force interact something and it–

JAIM:

One thing to fail a test. You shouldn't take down the whole test suite. That's a huge win right there. What does that assert in quick?

NATASHA:

You would just say like, "Expect my string dot to equal five" So it's just a normal expect statement. It just does it automatically for you.

JAIM:

But behind the scenes it's checking the optional or if there's a value.

NATASHA:

Yeah.

JAIM:

Doing the right thing. Very cool.

CHUCK:

Do you find that a lot of people are trying to test their Swift code? Or is there not such a strong culture of testing in the Swift in ecosystem yet?

NATASHA:

I think it's more of an iOS thing not specifically Swift. So it just depends. I think the community is moving definitely a lot more towards testing. But I'm sure there's a lot of people who don't test so. It's hard right? Because you have to put the time upfront. So just something you have to trust in because upfront, it really slows down your process.

ALONDO:

Yeah, guilty here of not doing it.

NATASHA:

But it's very addicting once you do it because you have the security blanket and you can – I personally, my happy day at work is deleting a thousand lines of code.

CHUCK:

Oh, yeah.

NATASHA:

When you don't have any tests, it's like, “Well I hope nothing uses this old thing anymore. The code seems to compile.” But when you have the test, you just have a lot more confidence deleting the old code.

JAIM:

If you like you should have out a test on it. [Chuckling]

24:

53]. I was trying to merge back to something and I ran the test and they broke. They found it out very quickly versus doing massive Git refactoring. It can help you save a lot of time trying to figure out what went wrong, especially if you’re a crazy rebase.

NATASHA:

It also acts as a documentation for your code especially if you’re writing quick style test where you're saying, “This is the context. This is what I'm expecting for each case.” It becomes a lot readable and someone looking at your code if they're not sure, they can go to the test be like, “Okay, here's what's expected of this method.”

CHUCK:

You get all of the – like you said, the setup and everything else so it's like if I want to just open up some sandbox a playground or something else and just fiddle with stuff, then I have all of the setup. I know exactly how to put things together so that it will behave the way that I expect it to.

NATASHA:

Exactly.

JAIM:

So what are some other differences when we’re testing between Swift and Objective-C?

NATASHA:

I think the main one that’s just an annoying one where you have to put your each file into the test target for you to access it. Or the other work around is you have to import to your whole framework. So, just not automatic it's a little bit clunky to set up and then just dealing with optionals is painful sometimes. But that's just more Swift versus – not even testing specific.

JAIM:

Yeah, I think you highlighted the first major battle in the Swift TD wars. Whether to import every file or make everything public; what is the better approach?

NATASHA:

Yeah I could tell you what I do. I don't know which one is better but I like to import each file because I’m very obsessive with keeping things private. So – but it's not fun.

JAIM:

Yeah, definitely I think it – unfortunately, yes you alluded to that your test bundle can't see your internal methods, which is a pain, because we like to keep them internal not have to set everything. I tried importing all my files to the test project and I found if they had any dependencies on CocoaPods, I had to import everything. I walked away from that approach and I just make things public and just be careful of what I need to test. One of the interfaces that I’m exposing, which for the project working on, works pretty well. If I was doing an API where I had to watch the interface

very closely, I might think differently. But I've had less success importing into the bundle. A lot of people seem to like it.

NATASHA:

Yeah, hopefully future versions of Swift will make it a lot easier to do that.

ALONDO:

Yeah, so let us see the internal stuff. That would make things a lot easier.

CHUCK:

How much of this do you think is going to change over the next little while because they’re still playing with some of the tooling and stuff around Swift?

NATASHA:

Yeah, it’s pretty – it’s become more stable. I think as Apple’s been focusing on WatchKit but I think we'll probably see some kind of Swift 2.0 at WWC. So, at that point it’ll probably have breaking changes. It shouldn’t be as fun, but it's hard to predict.

CHUCK:

Yeah, that’s true. It’s definitely hard to know exactly what they'll do.

ALONDO:

It seems there’s a good bit of civility – I’m hearing lots of people doing development on Swift saying they don't seem to be – seems a bit of confidence that at least at this stage it's good enough to move forward with projects.

NATASHA:

Yeah, definitely. And, again, it hasn't been – they do releases every two weeks for the new beta and it hasn't had anything breaking. I think they're probably focus more on optimizing it, making it faster, things like that.

JAIM:

Less beach balls.

NATASHA:

Yep.

JAIM:

Crash! I typed in something bad and wrong! Crash!

NATASHA:

Yeah Xcode crashes. [Crosstalk 28:42]

JAIM:

How many ways can you crash the compiler?

NATASHA:

Infinite. They fixed the terminating the source kit stuff a little bit. It’s much better now.

JAIM:

That’s good.

NATASHA:

That was very painful for a while.

JAIM:

Okay. I’m on Beta two so I need to upgrade. [Inaudible 28:58]. I heard four was bad. What are the problems with four? You have an opinion?

NATASHA:

Well at work, I use a stable version; I don’t use the betas. A home, I've been using betas. They seem fine but I’ve been focused more on WatchKit stuff not as much Swift specific.

JAIM:

Okay.

CHUCK:

So I'm a little curious. And this is – I'm going to go wildly the off topic here. But so what's your process for experimenting with Swift and then writing a blog post about it?

NATASHA:

Usually, it’s just something I do on a daily basis. So it's more like I'm just coding like a normal person. I go to work, code or work on my own project and I’m coding something and then I get stuck. And then try to figure it out or sometimes I read blog posts that give me ideas of how to do better code. So, one example is I’ve been reading a lot about functional programming because – and that has helped me improve my Swift code and make it more Swifty. So it’s a mix of learning and doing.

CHUCK:

I was hoping you were going to walk into work, steeple your fingers together and then say something to the effect of “I am going to hurt Swift today.” But mostly it's just around the “Okay, this didn't do what I expected. So what does it mean?”

NATASHA:

Yeah, or sometimes I just read a blog post and I’m like, “Oh! There’s a better way to do this!” As one example is – I don’t know if you’ve seen the blog post of like error handling and Swift. I think it’s very elegant and it takes from the functional programming world where you can have an enum where you have an error – like a result enum and you have a success and error case. And then you have switch statements based on whether you're – you have a success or error result. But that type of stuff, I read and the go to work and I want to implement because it's so beautiful. And it’s more Swifty but it's not something that I would have come up with from what I'm used to doing in Objective-C.

CHUCK:

Gotcha! So you're experimenting at work. You’re experimenting at home on your own time. Just reading what strikes your fancy.

NATASHA:

Yeah, read a bunch of stuff. See what people who are smarter than me – what they do. And feel the comfort zone because, for example, like a lot of – there’s some people out there who are like

super crazy with operators in Swift and they have ten different operators. And – to me, that's cool to look at but that's not as readable, or like people on my team will know what this means. So, I'm not going to be implementing that but it's interesting to read about.

JAIM:

I think people that don't know C++ are doomed to repeat it.

ALONDO:

Don't do it people!

JAIM:

But it is – operator overloading is not quite what it is in Swift but a little safer than it was C++ like I couldn't see some use for it but take it easy don't do it too much.

ALONDO:

I had one question about something you mentioned earlier I the show about unwrapping multiple optionals. So the brute force way is just doing if-lets and then inside that another if-let and that's annoying for everyone involved. How do you do it?

NATASHA:

So you can just do a case statement. So pattern matching and then you just do case, or switch, and maybe have two things and say like, "first optional comma second optional." And then you're doing switch statements and each case is going to be like if both of them are there; if one of them is there; if the other one is there; if none of them are there.

So it really forces you to think of every single scenario not just like – because if you do the nested if's, you're only testing the success scenario if both things are there but by doing the switch statement, you have to think of every single combination of both there, first one there, second one there, none there.

JAIM:

Okay, so pattern matching – that's a paradigm from functional programming. Is that the same that you would do in like a list similar to it? I hear the term but I don't really fully understand it. But just doing a case to switch in a case statement where things that you expect to happen, happen?

NATASHA:

Yeah just basically a case statement that says these – I expert 1, 2, 3 like it falls through. It's – I don't think and – I think maybe the word is too complicated, but it's just a pretty simple switch statement with fall through cases.

JAIM:

Yeah, I tried doing that and I ended up crashing the compiler.

NATASHA:

Oh, awesome! [Laughter] [Crosstalk 33: 27]

JAIM:

If I can fix it, then I'll try again but maybe just had some syntax wrong. I can fix it. But definitely a better way because I've been nesting my if-lets and it's annoying.

CHUCK:

Yeah, if you want to do pattern matching where it's like the way things are done then error laying and elixir really do that as far as other languages go and yeah, it is a very functional concept. Well, I'm bookmarking your blog.

NATASHA:

Thanks.

ALONDO:

I'm actually reading the post now.

NATASHA:

Is that the multiple optionals one?

ALONDO:

No, actually I'm farther behind so I’m actually reading the Unit Testing in Swift: A Quick Look in Quick Quick.

CHUCK:

Anything else we should go over? Anything exciting going on in your career or life you want to go through or should we go to picks?

NATASHA:

I'm excited about WatchKit!

CHUCK:

WatchKit.

NATASHA:

Yeah, the watch should be coming out in April. They actually announced it.

JAIM:

Are you getting a watch?

NATASHA:

Hopefully! I don’t know – the price points are a little – I think there's one that's like three hundred dollars so that one seems good. Yeah, so probably. It’s hard to develop for it now on the simulator because in real life, it’s going to be so different. So, I think we’ll definitely need it for that actual testing.

ALONDO:

I agree.

NATASHA:

You need to develop it for–

CHUCK:

It’s on my list. I can't convince my wife though that I need one.

NATASHA:

I, most of all, really need one. It’s your job.

CHUCK:

I'll just get you on tape saying that then should go look at me go, "Well, who the heck is she?"

So you're doing anything interesting right now with it or?

NATASHA:

Yeah I haven't had as much time as I'd like but mostly just playing with it – understanding. The hard part I'm trying to learn right now is – and you'll see a lot of blog posts coming up. Probably not this week but maybe like in two weeks, I'll do a lot of them. And it's really understanding how to share data between the watch and the iOS app. I'm not sure but like share the code and the data. So I'm learning better because you have to make your code slightly different.

CHUCK:

Makes sense. I'm really excited to play with it but I always have something going on.

NATASHA:

It's still very new and very limited. So, I think the hard an interesting part is architecting your full project in a way that makes it easy to work with the watch and your app. Like one example is I want to use frameworks for your model code. So, now you can use the same code in your watch app and your iOS app and maybe your Mac app. So, that's a good coding practice that I haven't really done but just something to a get used to.

CHUCK:

Alright! Well, let's go into some picks! Jaim, do you want to start us off with picks?

JAIM:

Sure! I got some picks. So, I got some blog posts. So, we talked about a couple ways to make our code accessible for unit tests. We can make everything public. It's more my approach – a lot of people are – this is a test bundle Saul Mora came up with a third approach, which requires deriving from an NS object and some run time stuff but there's a third way of doing it so I thought it was a pretty cool article. So, "Unit Testing in Swift Without Access Levels." So, if you're developing APIs and you want to keep them locked down in private and still test them, it's an interesting approach. So, initially I was turned off by using Objective-C stuff in Swift but I'm becoming more open to it because it's happening all the time anyway. If you're doing any framework stuff, you're coming from NS object – that's one thing.

I also noticed – we talked about dependency injection earlier in the show. And also my blog – couple links from Objective-C – objc.io. John Reed did a great post on dependency injection so if we throw that word out you're not sure what it is, it's good explanation of it. So, I'll put a link to that.

And a shameless pick. I did a blog earlier this week about Testing Singletons in Swift using some constructor injection. So, those are my picks. Three blog posts if you want to learn more about testing.

CHUCK:

Alright! Alondo, what are your picks?

ALONDO:

I have two picks this week. The first one is a blog post as well on – we've been a – we're updating our API and we're looking at parsing collection JSON. And so one of the techniques we're investigating this week is using the Cocoa's KVC to do the parsing and looking at some ways that we can handle that so it will be easy for us to streamline the code reduce – and delete lines of

code. Sans testing but we will try to get there as well. So, it's a pretty interesting technique and I think it's something that we will use regardless of the specifics of our particular implementation.

The second one is not technical. It's a budgeting app. I started trying to work through the Dave Ramsey's seven steps. I know you've turned me onto a couple of things, Charles. And so I started using a software program called "You need a budget" and I find it really helpful. I was new to the whole idea of zero-based budgeting. So, it's helped a ton. So much, in fact, that I'm actually moving forward on steps and I'm super excited. So that's my second pick and that's all I have this week.

CHUCK:

Awesome! I have been pretty much in the throes of putting up a conference for JavaScript developers and that is this week and next week so I've been pretty tired. I don't have any picks per se, though I do have one thing that I am pulling together. I'm going to get the landing page up this afternoon. And that is for DevBoxClub.com. And what it is it's a subscription service and I'll just pack a bunch of stuff into a box and ship it out every month. I had a lot of people basically say books, T-shirts, desk keys, different things related to different software products. I have been talking to a lot of folks and as you can imagine with five shows that I do and all the people I talk to, that I have a lot of connections that I can use to pull the stuff together. So, if you're interested in getting a box sent out to you every month, I'm still working out the pricing and stuff, you can go to Devboxclub.com and sign up for the mailing list. And I will let you know when you can sign up and how much it cost per month to get a box.

The other thing that I want to say about that is that I've decided that the first run is only going to be 50 boxes. And the reason is that way, I can figure out what the logistics are in sending out all the boxes and then I'll open it up to more people after that because then I'll know how much work it's going to be to put them together put them out. So, if you want to get that early access to know when and where to go sign up because I figure fifty boxes is going will go pretty fast, then you need to be on the mailing list. So, once again that's devboxclub.com. And yeah, that's all I've got this week.

Natasha, what are your picks?

NATASHA:

Since we talked about testing, one of the testing frameworks written by the guys who did Quick is Fox. And Fox is a property-based testing. It's based on that kind of quick check and has cool other functional languages. And that does a lot of auto generated input so it's very thorough. So it's fun to check out.

Another one is also from Objective-C.io is that they have a Functional Swift book. So that book is really helpful for understanding the functional aspects of Swift and deciding where you want to fall on that.

And then, David Smith, he's been writing a series called "As I Learn WatchKit" and that's been really great to just – he has videos and blog posts of just all the WatchKit stuff. So, I highly recommend it.

CHUCK:

Awesome! Well, thanks for coming and thanks for sharing your expertise. I think this is an area that more and more people are going to be needing to learn so if people want to follow up with you, what are the best ways for people to know what you're up to?

NATASHA:

So probably, Twitter. It's @NatashaTheRobot. I have an RSS feed on my blog. I have a newsletter. So, lots of ways.

CHUCK:

Awesome! Alright! Well, thanks for coming. We'll wrap the show. We'll catch you all next week.

[This episode is sponsored by MadGlory. You've been building software for a long time and sometimes it gets a little overwhelming. Work piles up, hiring sucks and it's hard to get projects out the door. Check out MadGlory. They're a small shop with experience shipping big products. They're smart, dedicated, will augment your team and work as hard as you do. Find them online at MadGlory.com or on Twitter @MadGlory.]

[Hosting and bandwidth provided by the Blue Box Group. Check them out at BlueBox.net.]

[Bandwidth for this segment is provided by CacheFly, the world's fastest CDN. Deliver your content fast with CacheFly. Visit cachefly.com to learn more]

[Would you like to join a conversation with the iPhreaks and their guests? Want to support the show? We have a forum that allows you to join the conversation and support the show at the same time. You can sign up at iphreaksshow.com/forum]