iOS Debugging & Instruments

Overview

- Alternatives to Debugging
  - Print statements
  - Asserts
- Important xCode Settings

- Debugger Tour
- Instruments: Allocations/Time Profiler
- Debugging Advice
- Practice Debugging
- Essential Tools

What I don’t cover

- This is just an intro.
- I don’t cover LLDB commands.
- The debugger has a whole CL interface that is very powerful.
- My advice: Put your energy into writing good unit tests instead of spending your life learning LLDB commands, but sometimes we don't have a choice!

NSLog/Print

- Some log tricks

```

NSLog(@"%s", __PRETTY_FUNCTION__);

```

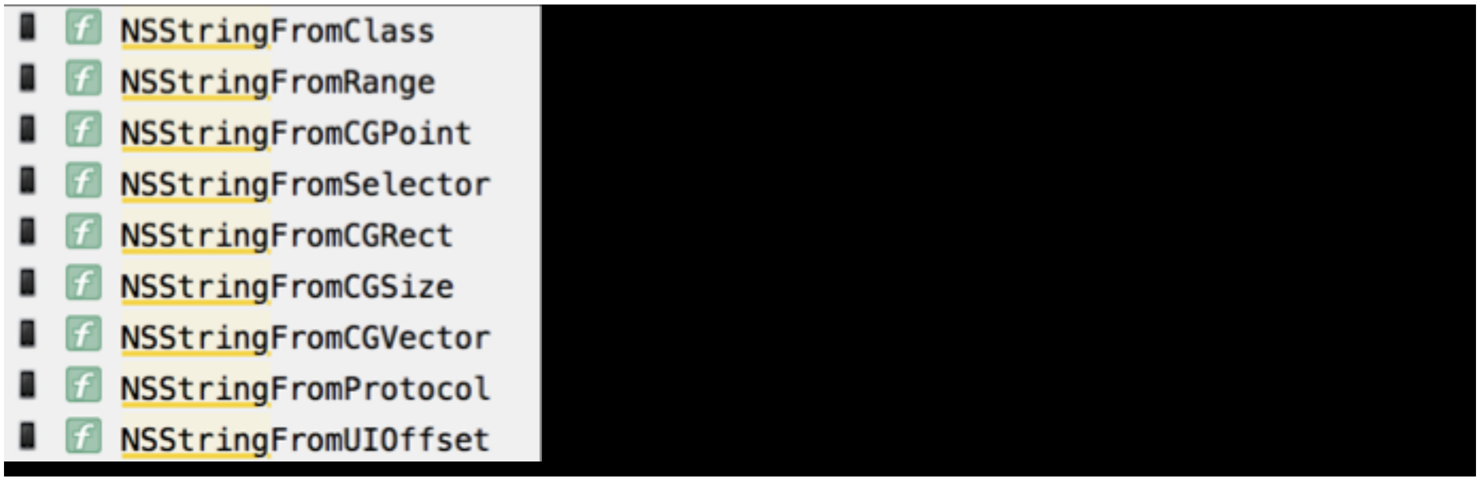
```
/*
#file
#function
#line
```

```
#column
*/

print(#file, #function, #line, #column)


```

- Convenience methods from converting to NSString (handy for doing print statements and other things in Objc)



```
NSStringFromClass
NSStringFromRange
NSStringFromCGPoint
NSStringFromSelector
NSStringFromCGRect
NSStringFromCGSize
NSStringFromCGVector
NSStringFromProtocol
NSStringFromUIOffset
```

```

CGRect rect = CGRectMake(0, 0, 100, 100);
NSLog(@"%@", NSStringFromCGRect(rect));

// Swift
print(rect)

```

Good/Bad of Print Statements

- Good
    - Easy, immediate, essential

- Bad
    - Called “cowboy debugging” for reason
    - Can introduce bugs
    - Need to be removed before shipping
    - DLog/ALog & other alternatives automatically removed from release builds
    - Makes code harder to read
    - If you get lazy and “forget” to remove your print statements you have the busy console problem

```
print("======>>>>>>!!!!!! HEY !!!!!!!
<<<<<<=========")
```

NSAssert/Assert

- We’ve seen Asserts in the tests exercise (eg. XCTestAssertNil())
- Asserts are functions that take 2 parameters.
    - The first parameter is some statement that is being asserted to be true.
    - The second, optional parameter, is a message that is logged if the assertion fails.
- Assertions assert something to be true, and if that statement is not true the app crashes and dumps the message to the console.
- Very handy for development debugging.

```
// Objc
NSAssert(self.data, @"data should not be nil");
NSAssert(self.data.count == 20, @"data count should
be 20);
```

```

```

```swift
// Swift
let num = 10
assert(num == 10, "This message will not run because num is 10")
assert(num == 11, "The app crashed because num is not 11")

```

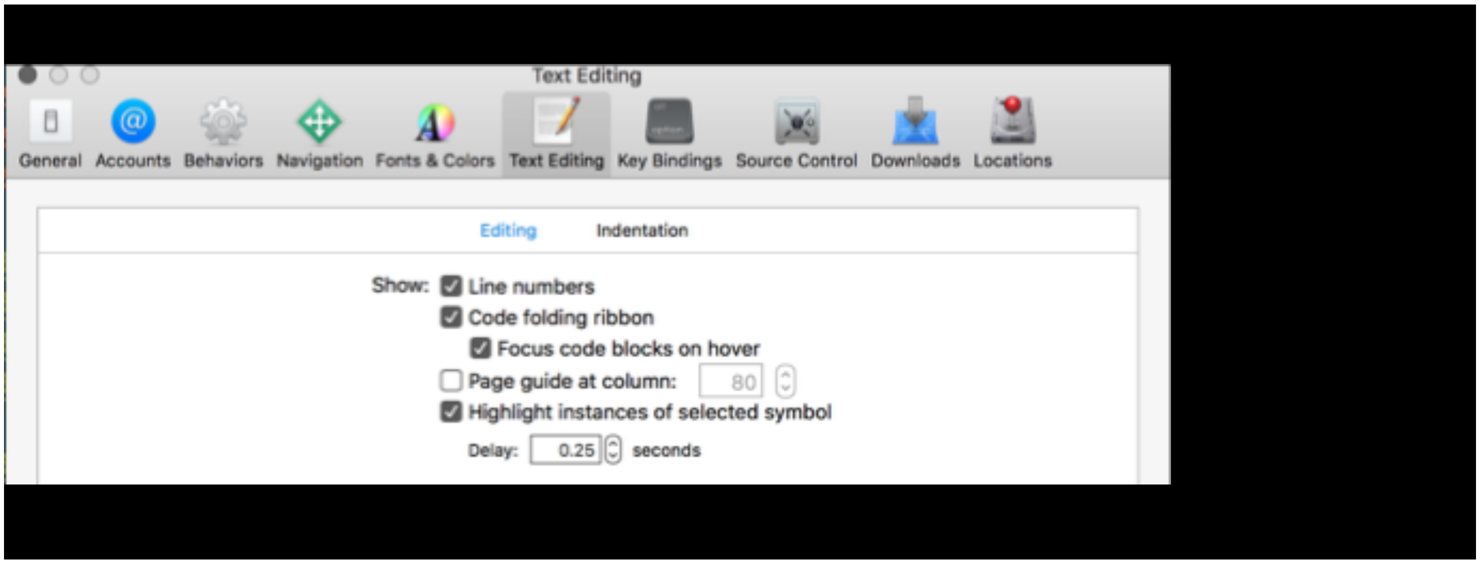
- Question
  - Why would you want your app to crash if some condition isn't met?

- Problem with Asserts
  - They should be removed from production code & "someone" might "forget" to remove asserts from production code.
  - But you can use macros that automatically remove them from production code (eg. ZAssert).
  - You're adding code to your *app* target to do testing which can introduce bugs.
  - Might as well write unit tests instead! These are afterall asserts, but they are in a target separate from your code. Much smarter. UNIT TESTS == BETTER.
  - But for quick tests in an app that isn't using unit tests, it's a reasonable choice.

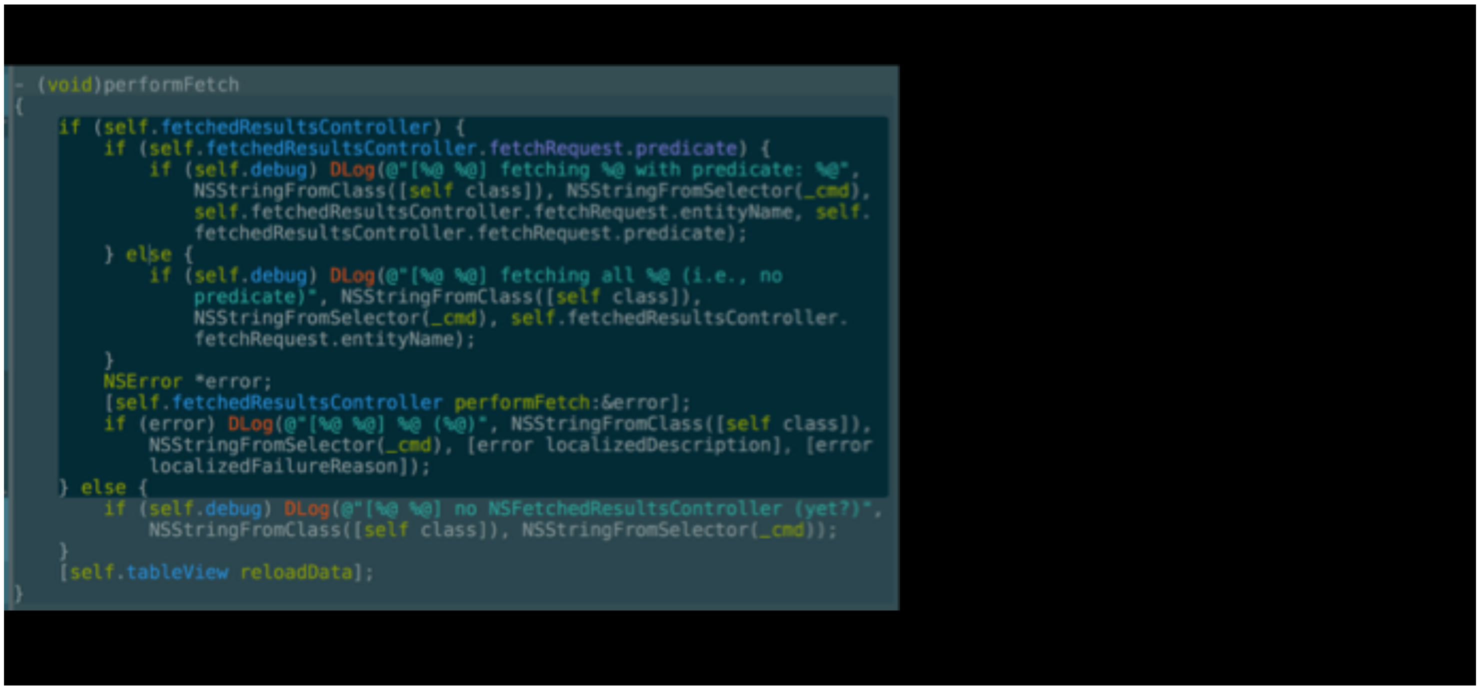
Helpful xCode Pro Settings

Folding Ribbon

- Make sure you enable the folding ribbon in Xcode.
- Great for solving scope issues.
- BTW, it’s most likely a "code smell" if you have to use the ribbon to figure out your scopes. Repeated if/else statements or switches ARE a definite code smell. What do I mean by this?



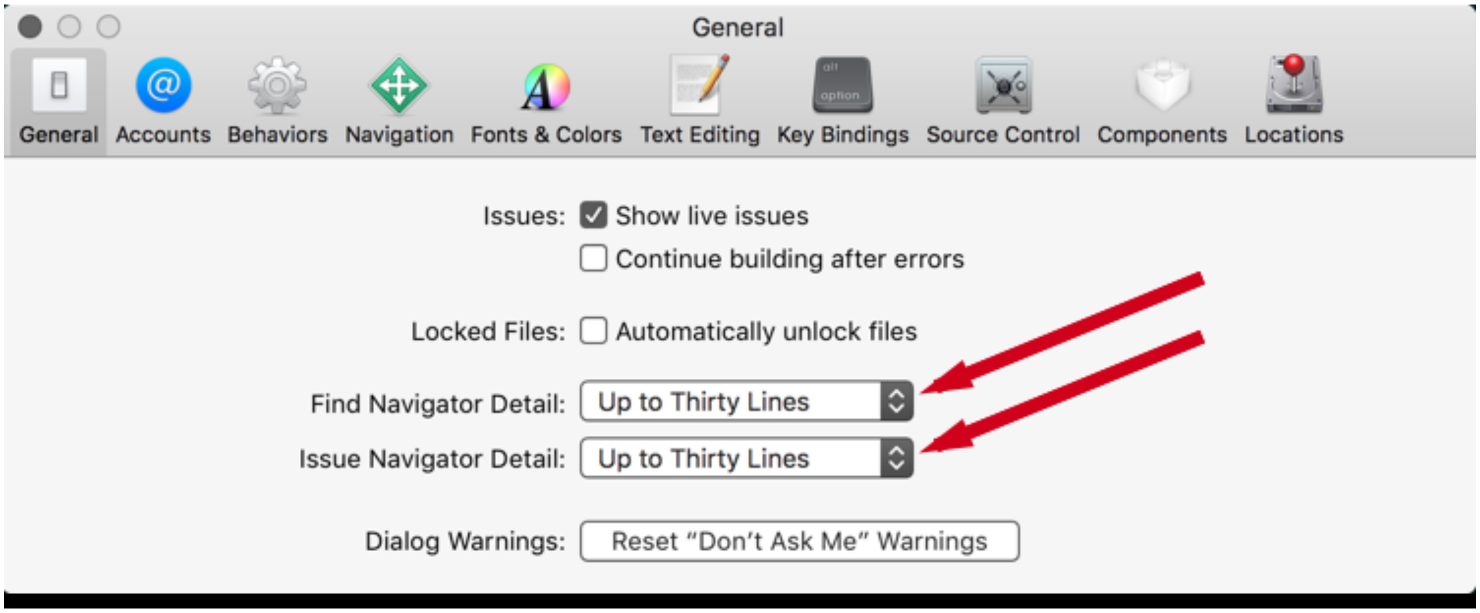
Show: [✓ Line numbers]
[✓ Code folding ribbon]
[✓ Focus code blocks on hover]
Page guide at column: 80
[✓ Highlight instances of selected symbol]
Delay: 0.25 seconds



```
- (void)performFetch
{
    if (self.fetchedResultsController) {
        if (self.fetchedResultsController.fetchRequest.predicate) {
            if (self.debug) DLog(@"[%@ %@] fetching %@ with predicate: %@",
                NSStringFromClass([self class]), NSStringFromSelector(_cmd),
                self.fetchedResultsController.fetchRequest.entityName, self.
                fetchedResultsController.fetchRequest.predicate);
        } else {
            if (self.debug) DLog(@"[%@ %@] fetching all %@ (i.e., no
                predicate)", NSStringFromClass([self class]),
                NSStringFromSelector(_cmd), self.fetchedResultsController.
                fetchRequest.entityName);
        }
        NSError *error;
        [self.fetchedResultsController performFetch:&error];
        if (error) DLog(@"[%@ %@] %@ (%@)", NSStringFromClass([self class]),
            NSStringFromSelector(_cmd), [error localizedDescription], [error
            localizedFailureReason]);
    } else {
        if (self.debug) DLog(@"[%@ %@] no NSFetchedResultsController (yet?)",
            NSStringFromClass([self class]), NSStringFromSelector(_cmd));
    }
    [self.tableView reloadData];
}
```

Show Full Error Messages in Sidebar

- Settings > General in Xcode, increase number of lines for errors!



Issues: [✓] Show live issues
[ ] Continue building after errors

Locked Files: [ ] Automatically unlock files

Find Navigator Detail: Up to Thirty Lines

Issue Navigator Detail: Up to Thirty Lines

Dialog Warnings: Reset "Don't Ask Me" Warnings

Debugger

- ==>> Debugger Demo: Open **BreakPointsTourSwift <<==**

Instruments

- Xcode has a massive instruments feature used for debugging and performance tuning.
- We'll just look very briefly at two of the most useful instruments
  - Allocations: takes a snapshot of all of the objects your app allocates, retains and releases.

  - Time Profiler: gives you data on how long your app is spending running various methods.
- ==>> Instruments Demo (Open: **AllocationsTest** & **TimeProfiler**) <<==

Debugging Strategies

- Avoid stabbing in the dark. THINK before changing anything.
- My Technique:
  - Describe problem thoroughly. Try to describe the precise conditions that trigger **unexpected** behaviour. If you need more info, gather it. THINK, don’t just start stabbing into the dark (i.e. commenting out lines **superstitiously**).
  - Form an hypothesis. Start with most obvious and easy to test.
  - Test your hypothesis.
  - If that isn’t it, go to the next most obvious cause.
  - Repeat until you find the problem and solve it.
  - Document your results in a Solutions Log (Agile Best Practice).
  - Always take any compiler errors seriously. Decrypt them first.
  - Get in the habit of solving problems yourself before looking them up on SO.
  - Consider that a problem might have more than a single cause.
  - Avoid complex problems by a practice of continuous testing.

    - When building always try to get your code to a testable state, test and then move to building the next element.
    - Learn to write unit tests.

Debugger Exercise

- Let’s work on the debugger exercise
- Open ==>> **DebuggingExerciseSwift** <<==

Some Important Tools

Viewing Diff Files:

- SourceTree: https://www.sourcetreeapp.com
- P4Merge: https://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools

Networking:

- Paw: https://itunes.apple.com/ca/app/paw-http-rest-client/id584653203?mt=12
- Postman: Chrome Extension
- Charles Proxy

References

- https://developer.apple.com/support/debugging/

- https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/debugging_tools.html
- https://developer.apple.com/library/tvos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/
- http://lldb.llvm.org
- http://jeffreysambells.com/2014/01/14/using-breakpoints-in-xcode