Beyond Bootcamp

Me: before Detroit Labs (Jan 2014)

- ☞ Small, procedural programs (~100-200 lines; 1 file).
- ☞ Limited experience with IDEs (Xcode).
- ☞ No experience with version control (git).
- ☞ **No idea how to bridge the gap** between my projects and "big", "professional" projects.

Me: now

☞ Large, event-driven apps (~100k-200k lines; 1000+ files).

☞ Shipping Android & iOS apps to 100k+ users.

Goals of this talk

① Explain **why** professional app development is different.

② Explain **how** professional app development is different.

③ Explain what those differences will mean to **you** as an individual developer.

④ Provide examples and resources to help you bridge the gap.

⑤ (Some iOS specifics.)

Caveats

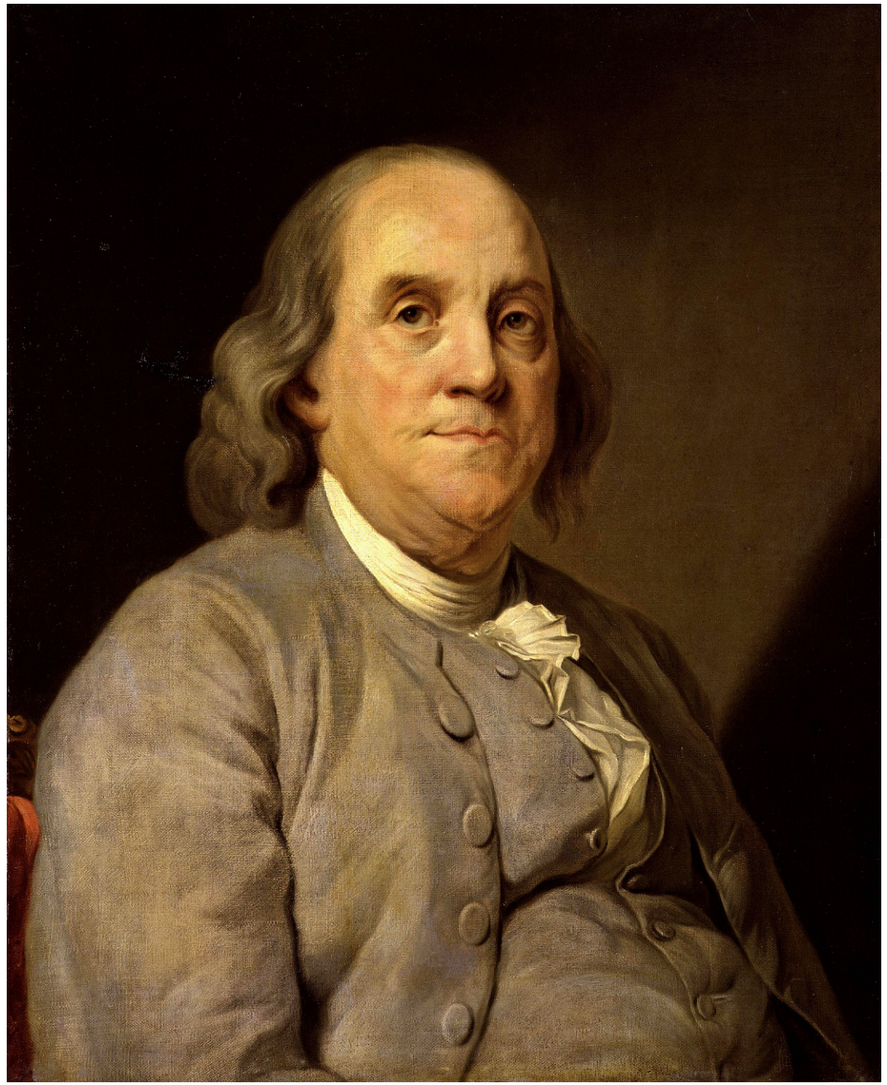

YEAH?
WELL, YOU KNOW, THAT'S JUST LIKE, YOUR OPINION, MAN!
quickmeme.com

The **why**

The constraints

“Time is money”

—Benjamin Franklin



“Perfect is the enemy of good”

—Voltaire

The compromises

The **how**

What do these constraints mean for **companies**?

① Customer timelines → developers work in **teams**.

② Formal training is (doubly) costly → hire **adaptable** developers who can **learn on the job** OR hire **experts**.

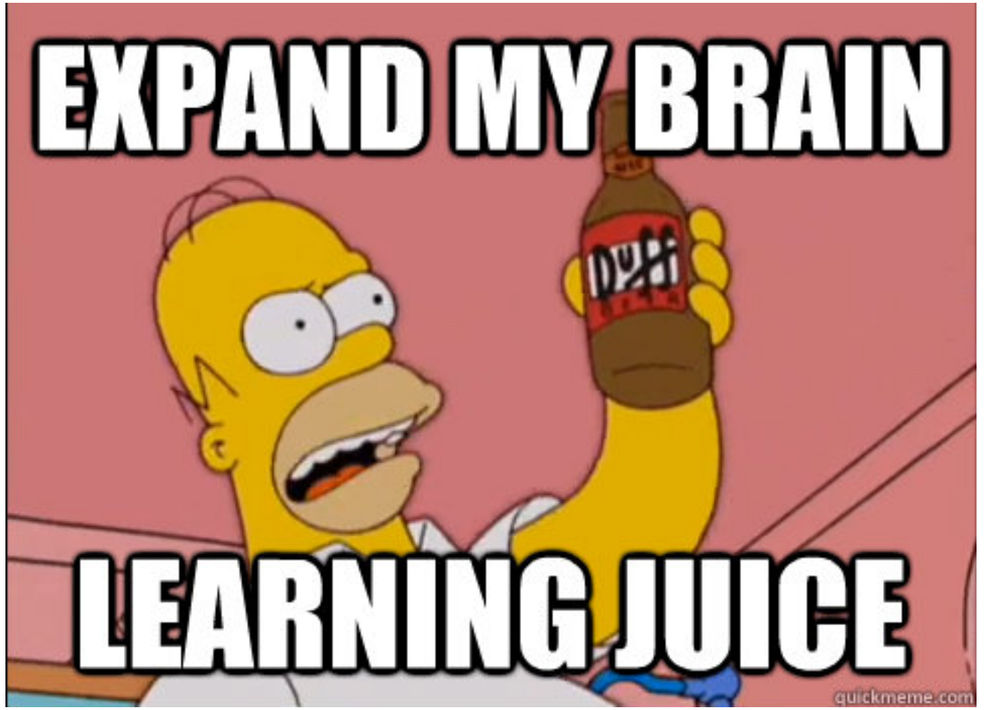
What do these constraints mean for **you, as a developer**?

Capacity to contribute and grow as a pro developer is based on:

① Your **communication** skills.

② Your **self-directed learning** skills.


MEMES
EFFECTIVE COMMUNICATION, THEY ARE
memes.com


EXPAND MY BRAIN
LEARNING JUICE
quickmeme.com

Overview: **communication**

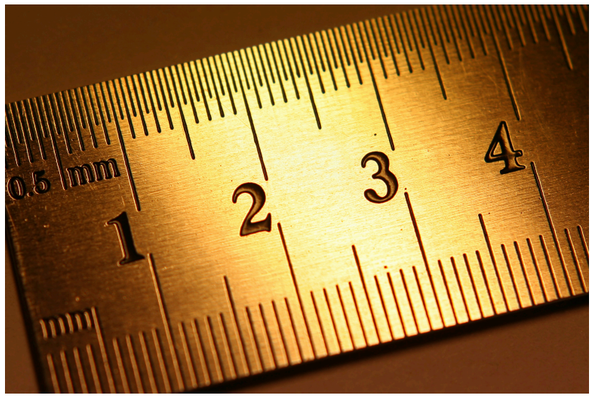
① With colleagues.

② With clients & customers.

③ In your code.

④ Around your code.

Communicating with colleagues



Resources

☞ [book] Thanks for the Feedback

☞ [book] Soft Skills: The software developer's life manual

Communicating with clients

Goals

☞ "Basic hygiene" (capitalization; spelling; grammar; phrasing).

☞ Audience-appropriate (e.g. developer vs decision-maker).

Resources

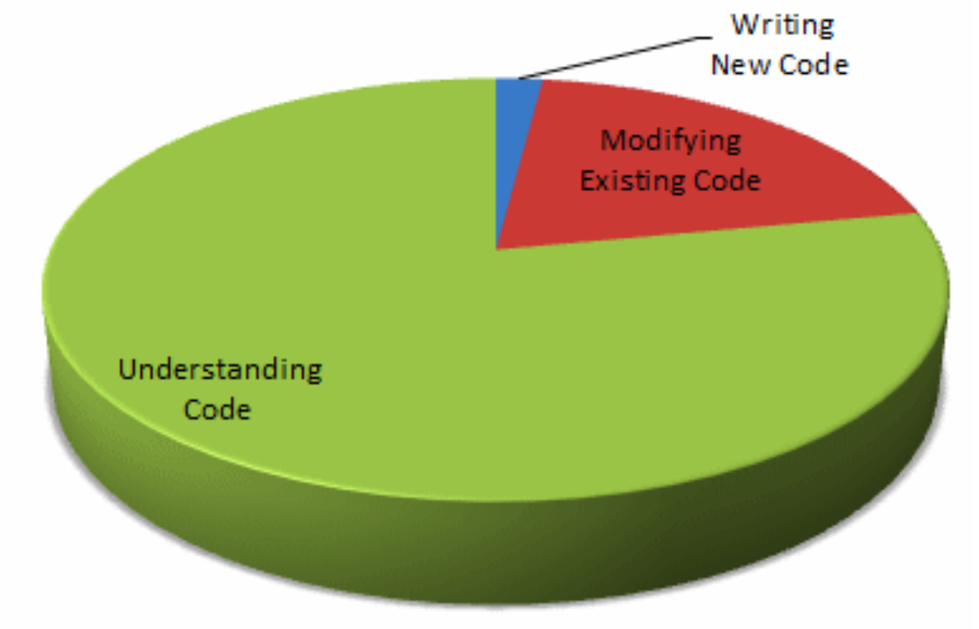
☞ [book] The Elements of Style

☞ [book] Soft Skills: The software developer's life manual (again)

Communicating in code: **why?**

“New code
becomes old code
almost instantly”

—Peter Hallam, Microsoft



Communicating in code

“A huge amount of what a developer is doing is in their head. As we write code we need to keep a mental model of how parts of the application that have already been written and are yet to be written interact with the part that we are writing at that moment. We need to have a solid picture of exactly how the code is working as a whole and maintain that picture. It’s not easy, it requires a lot of concentration and has to remain in place while we think of creative and efficient ways to solve the problem at hand.”

—Derek Johnson

Question

What does this line of code communicate?

```
var x = -17.7778;
```

☞ Some other code consumes the value represented by `x`.

☞ The initial (perhaps default?) value of `x` is `-17.7778`.

☞ The value represented by `x` may vary.

☞ The writer was probably an Objective-C programmer!

Question

What does this line of code **not** communicate?

```
var x = -17.7778;
```

☞ The value **x** represents (a length? a dollar amount?).

☞ The meaning of the assigned value (initial? default?)

☞ Whether the assigned value is rounded or exact.

☞ The intended scope of **x**.

Beware of **communication by omission!**

Communicate intentionally

```
public let zeroDegreesFahrenheitInCelcius = -(160.0 / 9.0)
```

☞ Scope explicitly indicated.

☞ Constant nature explicitly indicated.

☞ Value represented (and its units) explicitly indicated.

☞ More accurate initial value used.

Communicate intentionally

Always choose the appropriate:

- ☞ Scope
- ☞ Mutability
- ☞ Name
- ☞ Length (usually, short!)
- ☞ Location (DRY; SRP)

“Leave this world a little better than you found it”

—Robert Baden-Powell

Communicating around code: **why?**

Sometimes we need to provide **context**:

☞ explain **why** code exists.

☞ explain **when** methods should be called (if not obvious).

Note: code is still responsible for communicating **how** it works!

Communicating around code: **how?**

Three main options:

① inline comments

② tests

③ ~~formal documentation~~ (separated spatially; hard to maintain)

Inline Comments

Example of a **bad** comment, used as a crutch:

```
// This method applies a fix that prevents a crash.
private func someObscureName() {
  // Non-communicative and overly-long code
}
```

Inline Comments

Example of a **good** comment, used to provide context:

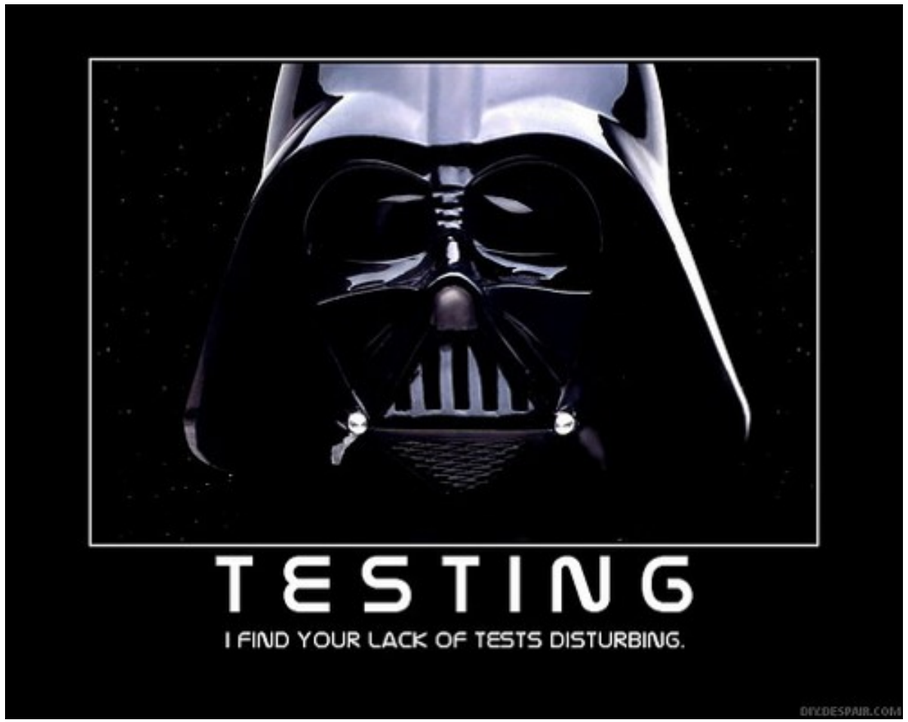
```
// This method fixes a crash introduced in v3.1.0.
// It should always be called during application startup.
// This code will be safe to remove once we update to v4.0.0.
private func applyCrashFix() {
  // Communicative and succinct code, calling into several
  // shorter subroutines if needed.
}
```

Tests: **why?**

Help us maintain **app stability while moving fast**.

Tests: **why?**

Act as **living documentation** of the code (forced to update).



Tests: stuff you should know

① There are different types for different levels of abstraction.

② Testing elicits strong opinions!

☞ **How much** is enough?

☞ **When** should you write your tests?

③ Keep test code clean.

④ Express **external requirements** through your tests.

Communicating in/around code: **summary**

- ☞ Write code with future readers in mind.
- ☞ Consider every inclusion/exclusion carefully.
- ☞ Use comments and tests to provide context.

Communicating in/around code: **resources**

- ☞ [book] Clean Code
- ☞ [book] The Pragmatic Programmer
- ☞ [book] Effective Unit Testing (Java)
- ☞ [blog] Martin Fowler
- ☞ [website, long form articles] objc.io

Overview: **self-directed learning**

① Short term

☞ Balance productivity with investment in learning.

② Long-term

☞ Address your weaknesses.

☞ Steer your career.

Defining self-directed learning

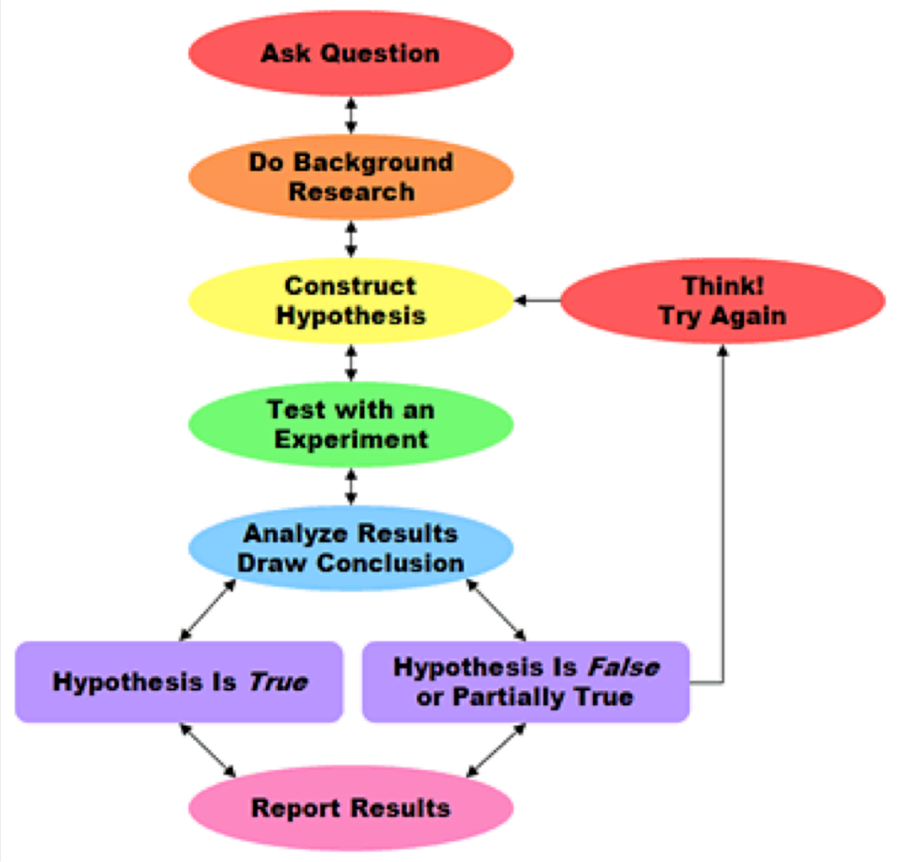
☞ Individuals take **initiative and responsibility** for learning.

☞ Individuals **manage** and **assess** their own learning activities.

☞ Mentors provide **framework** and **context**.

☞ Peers provide **collaboration**.

Measuring your knowledge



Measuring your knowledge

Hypothesis: "I understand (a class, language construct, etc.) X"

Tests:

① "I can reliably use X in my own code without looking it up"

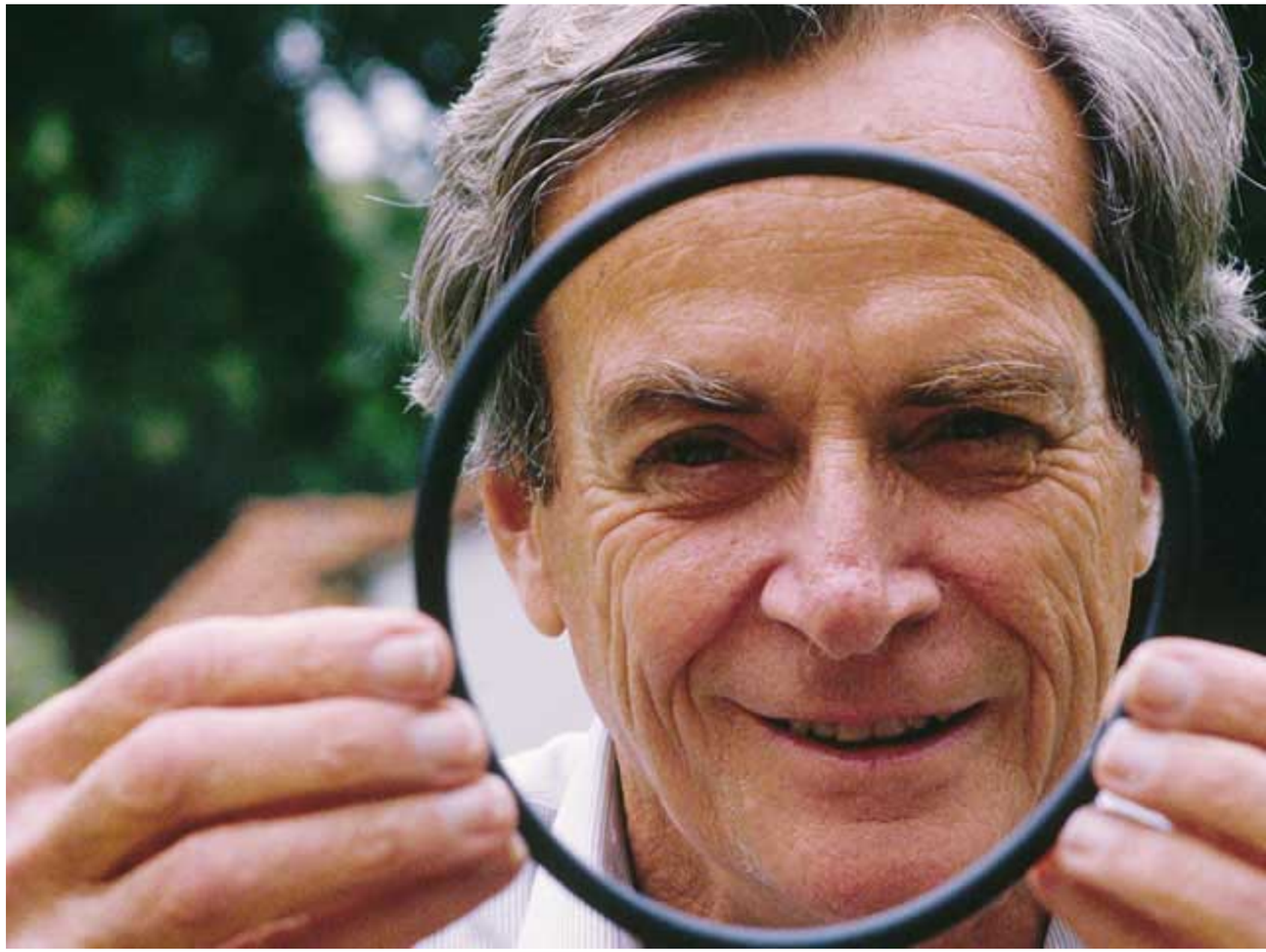
② "I can explain the details of how X works in general"

③ "I can place X in context and explain factors in its design"

④ "If I had to, I could write my own implementation of X"

“What I cannot create, I do not understand”

—Richard Feynman

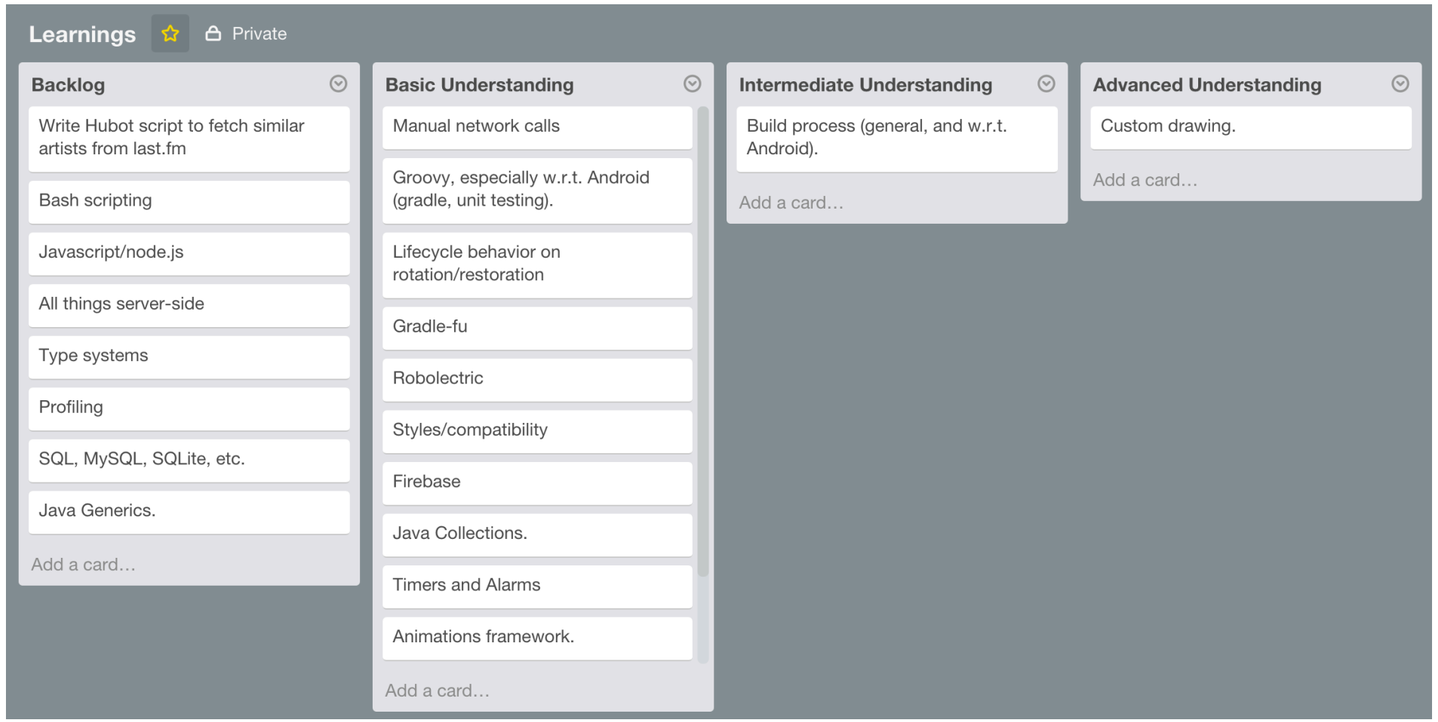


Tracking your progress

- ☞ Sense of accomplishment.
- ☞ Great for performance reviews/raises!
- ☞ Helps you gain a better sense of your overall strengths/ weaknesses and **plan** accordingly.

Tracking your progress: **how?**

I like using Trello. Easy prioritization.



Expanding your knowledge

Leveraging your client work

- ☞ If you can, **request** the tasks that will let you learn things you want to/need to know, even if you're slower at them.
- ☞ **Pair** with experienced developers on cards.
- ☞ **Observe** experienced developers, asking questions as you go (more available brain power than standard pairing).

Leveraging your mentors

- ☞ Don't expect them to have time to teach you.
- ☞ Do expect them to be able to provide answers and context when you are stuck.

Leveraging your mentors

How to **ask** a question when you're stuck:

“Hi! I'm trying to accomplish [goal]. So far, I've tried [thing1] and [thing2] but neither of them worked. My current hypothesis is that [your current best guess], but that doesn't seem to make total sense because [some annoying obstruction]. What am I missing?”

Leveraging your mentors

How to **follow-up** when a mentor answers a question:

“That makes a lot of sense now, thanks! What were the clues that led you to this solution?”

Self-directed learning: **summary**

- ☞ Measure and track your progress.
- ☞ Try to assess your strengths and weaknesses scientifically.
- ☞ Utilize mentors' experience and high-level view of the world.

Self-directed learning:

resources

These are hard to find. I'd love any recommendations you have!

The well-rounded **iOS** developer

① Become comfortable with both Objective-C and Swift.

② Know how to write unit tests.

③ Learn some shell scripting (for building apps outside of Xcode).

④ Understand app distribution (non-trivial!).

Objective-C & Swift

☞ Swift is structurally influenced by Objective-C.

☞ iOS frameworks are still written in Objective-C.

Resources

☞ [book] Objective-C Programming (Big Nerd Ranch)

☞ [website, long form articles] objc.io (again)

☞ Apple docs (switch between Objective-C and Swift modes)

Unit tests

☞ XCTest (default) is a great place to start.

Resources

☞ Apple docs on testing basics

☞ [long form article] objc.io #15 (yet again)

Shell scripting

- ☞ Used when building iOS apps outside Xcode.
- ☞ Also insanely useful for a bunch of random tasks.
- ☞ So powerful it should come with a health warning →

App distribution

- ☞ Apple security requirements make this a pain.
- ☞ First step is to understand digital signing: [book] Understanding Cryptography
- ☞ Then understand the "flow of signatures": [blog post] Ray Wenderlich: iOS Code Signing: Under The Hood

The End

Questions?

GitHub: **stkent**

Twitter: **@skentphd**