README

Elm Commands

```
> elm --help

> elm init

> elm install elm/json

> elm make src/Folder/ModuleName.elm --output=00-folder-module-name.js

> elm reactor
```

The best way to learn (redux)

> *It really depends on how deep you want the learning to be. Is it something I can look up in the documentation later? Will that be clear enough for me to do the job? Do I need proper examples to help me in the future?*

1. Lazy loading of cards, then chunk/consolidate learning points
    - This is more time consuming with intermediate learning.
2. Isolate the learning point into a single file or module
3. Show it working *within* the program (with comments in-place)
4. For *some* learning points, add Anki Cards
5. It's sometimes helpful to write a summary book too.
6. Create some drawings, perhaps storify the learning point
    - Add some memorable examples that are your own.
7. Now, from time to time, refresh your knowledge
    - Watch a video, write an example, read an article, or another book on the subject. You've always got the documentation to fall back on too.
8. It's really all about practice. Memory will only get you so far.

There's often a lot packed in to a book, and you don't want to be rewriting the damn thing too much, as that's time consuming, but for *deep* learning, this seems to be the way to go. You've got Anki to drill your memory, you've got summary notes (or a book) for reference, and you've practiced a few examples to consolidate your learning.

The problem with Elm is sometimes there's documentation but not enough tutorials or examples of how to *use* these tools.

Do the simplest thing first!

> *Got a narly piece of `json` to decode? Start "as if" you've already imported it. Do the simple things first and leave the difficult stuff to the end (like catching errors).*
>
> ***Whiteboard that shit out first:* Spend 80% of your time thinking about how to code, then code with confidence!**

- Harcode your `Model`. Decide how data should look.
- Do you *really* need all that data? Can you simplify?
- Do we *really* need a `Maybe` type?
- Is `type Custom` a better choice?

Build out your functions "as-if" the data is already there. Once you're happy with the `Model` and the data it consumes, work backwords and test out the bits of the `json` you'll need.

- Is it always in the same shape?
- Are there some bits of data missing sometimes?
- Is there a better source for the data?

Someone says to put off things like `Maybe.withDefault` to the very end, and start with the data so that you get saner results. Do you need a `non-empty List`? Are some `json` objects fields optional? How are you going to validate the data that's coming from, or sending to the server?

The sad (but real) truth

> ***Do what you're good at.** I'll repeat. Do what you're good at!!! **You've got one life, so don't waste time.***
>
> *Learning to program (properly) is such a huge investment in time — just look how much one chapter covers!!! I only really want to be able to use it for simple prototypes, proof-of-concepts, until they're turning over a profit and I can find someone better than me to work with.*
>
> ***Remind yourself of that, and don't get bogged down with research for CompSci.** Only do it for fun or necessity (or for quick brain training).*

I've forgotten at least 60% of what I learned during HTDP (the college-level course) and recursive algorithms would probably take a good deal of rejigging in my mindbox to get right (they can get very difficult!). I don't think I have that kind of time at my disposal. There's lots of other things I'd rather be doing; high-level thought, a bit of design, learning the piano, writing a book; all of which are a considerable investments in time themselves.

- Some people make great developers
- Some people's mind is a finely-tuned logic machine
- Some people really *love* it

I'm not one of those people. Purely practical, or good to keep the grey matter alive. And that's OK. I'm happy to do simple forms, simple json, spreadsheets for a bit of data, AI for help, and GUI tools to do the rest.

No data base admin, no heavy lifting, no racking my brains for maths or difficult algorithms. Fuck that. I'll learn enough to teach, enough to get by, enough to make some money. I can follow documentation and tutorials, cobble together some basic websites, and that's enough.

- Some light `sql` would be fine but …
- Setting up a `postgres` db with a linux server is an effort …
- Learning the ropes of even light DBA stuff is time consuming.

The Structure and Interpretation of Computer Programs is a nice *theoretical* dream to complete, and is intellectually stimulating, but it's *such* an investment in time that would have diminishing returns. Furthermore, learning one language is a challenge but in most jobs you'll have to master multiple languages and *keep on learning* which can be trying. Things move especially fast in the javascript world.

It's a little like learning Mandarin — Don't do it!!! *Admire* the caligraphy, learn a few characters and stock phrases; move on to saner, better pursuits (for me!).

Timesinks

> *Get better at focusing on the essentials, and speeding up these tasks:*

- Writing notes on books or adding issues to Github
- Asking questions and figuring things out on forums
- Adding unecessary complexity …
- … Which I always end up simplifying anyway!
- Flip-flopping between tasks or learning goals
- Consuming too many books, articles, videos
    - Often trying to flip between them when I should full-ass one thing.

Things for Anki

> *Do you really need to remember how to do difficult recursive programs? NO!!! Mostly these are done for you with `map`, `filter`, `reduce` etc.*

- **Start with the simplest thing possible**

- Without practice you'll forget lots of stuff
- If you *really* want to become a good programmer, it's lots and lots of hours of practice.

For most of us who just want to think like a programmer, learning about data (`set` theory, `arrays`, subsitute functions for `arrays`, `index` lists, and so on)^, algorithms, simple functions (that could be ported to Excel), and problem solvers. It's helpful to know which data to use at any one time.

```
^ See elm-in-action folder 03, line 80 and folder 04, line 231.
  We use a `Random` function in two different ways, one with an
  Array, and one with a `Random` function in place of an Array.

  I think there also a way to use an indexed list, so you can do
  things like delete at an index. That's quite Pythonistic way.
```

To-Dos

There's a LOT packed into Chapter 4. It becomes a bit tricky to fit all that knowledge into cards.

*You **do not** need to consign all of this to memory. A good deal of it could be in a file with notes, or a small summary book of sorts. There's also documentation for things you're likely to forget…*

So, for some things, learn it silently. Add it to your Anki card but don't feel the need to explain it. Especially esoteric stuff like the unitless type `()`.

1. Basic data types: `list`, `record`, `tuple`, using the `Photo` record with `comments` and modifying in a tuple for `update`.
2. `Json.Decode` and `Json.Decode.Pipeline` visual explanation that a 12 year old would understand. You need to write this whole section in language that's easier to understand with imagery.
    - Each decoder passes it's information into one of the `Photo` decoder's record fields. **page 69** and **Beginning Elm** or other sources.
    - Show a basic decoder first (from `Json.Decode`) probably `map2`.
    - **Order matters:** the order you stack up the mini-decoders within our `photoDecoder` function should mirror our `Photo` constructor function's argument's order `<function> : Int -> String -> Int -> Photo`.
    - You can potentially order the decoders in any order, so long as they use the correct `"key"` and `data` types (such as `required "id" int`) but **beware that this will screw up our `Photo` model if they're ordered incorrectly.**
    - Remember `Photo` can be a curried function, or a partially applied function.
    - The decoder type signatures can be a bit confusing, but just picture it as passing your

`Photo` record down the line, with each mini-decoder filling in one of the fields.
    - Briefly explain `succeed` and how this works.
    - Also explain `hardcoded` and why it's handy (it allows us default values where they don't exist in the `json`).
3. Beginning Elm shows us how to set up our own localhost server. Provide a notes file with an example. Also see **pg 74**
    - But first, show an example of how to run a test on a mini-decoder, with a `"string \"which is escaped\""` and a `"""triple string"""` for our `photoDecoder`.
4. We need to be aware of the potential states of our program. Loading in from the server requires a `Result`. There's potentially an error there. To begin with, he doesn't bother writing this properly and simply uses a hardcoded data model (which represents what *would* be loaded).
5. Once the data is loaded, there's the possiblity that the json contains no photos. We could use a `Maybe Photo` for this eventuality, which gives us a `Just a` or `Nothing`. We now also need to change our `Model` to use a simple record which can store that maybe type.
    - Remember that you can't just use `model.photo` because it's now a `Maybe` type. We have to `case` on that type depending on if it's a `Just a` or `Nothing`. (see line 172 and don't do it this way)
    - We'll have to change our `update` functions as they must consume, unwrap, and wrap the `Just a` or `Nothing` fields.
    - Show all the areas in our app that needed to be updated to reflect this.
    - We can use `Maybe.map` to achieve this (rather than the more convoluted method). Show both options.
    - **Show a few examples of where `Maybe` is not the best option, and where a `type Custom` is a better choice!**
6. You can also make use of `Maybe.withDefault` when converting a `.field` into a `text "string"` (or another data type). See here for some examples.
    - If you find yourself peppering everything with `Maybe.withDefault` it's an indication you should probably be using a `type Custom` instead (although I don't have many examples of this refactor).
7. It seems to be good practice to only consume the data that you *really* need in your functions (especially `update` and `view` helper functions). Simplify wherever possible and **only consume the types that you must**.
    - An example would be instead of the whole `Model` you only need `Photo.url`, which is a `String`, so your type signature would be `String -> Html Msg` in your function.
    - **Revisit some examples in HTDP and `Racket Lang`** where you're using the rule of splitting functions into simpler functions; abstracting where needed.
8. Pay close attention to the `comments` form entry and button. Rewrite the following so it's simpler to understand:
    - `onInput : (String -> msg) -> Attribute msg` within a form, takes a function that returns a `msg` type variable. So `UpdateComment String` is a function and also a `Msg` type. The DOM event handler will pass the `event.target.value` as a `String` argument. Every time the value changes in the input field. See (3) in the `viewComments` function. (see also `(7)`).
    - Boolean for `disabled` (examples) on form elements.
    - Default `value` for form elements (~~automatically updates the model element passed to it.~~

It's an opaque type so you can't "get" the value)
    - `onInput` Html event (pass to a `Update String` message)
9. We don't have to worry about comparing too much with javascript, but it's worth noting that Elm's aim is to avoid side effects (mutating state).
10. HTML5 forms offer a nice addition by validating user entry, such as the `email` form field. **Careful! This can be manipulated by bad actors.**
    - You will allways still have to validate any data submitted on the server, making sure is clean and safe data. The required attribute can be manipulated by a malicious user.
    - Perhaps you could leave the validation to HTML5 forms, but make sure it's a `String` before saving it as `json`?
11. You don't need to add this to an Anki card as it's hard to remember! But the `()` unitless type is used for `Browser.element` and also requires `Sub.none` and `Cmd.none`. It also wraps the model and the command in a tuple: `( model, Cmd.none )`.
12. Have a little go at `map`, `filter`, `reduce` in Elm lang.
13. For simple `String` you can `(List.map (\_ -> li [] [ text _ ]) entryList)` so long as it's wrapped in a `ul []` — you *could* output a different value (such as `text ""`) if `[]` empty list, but you don't have to (there'll just be an empty `li` item)

To-Dos (perhaps)

(2) and (3) have been deleted from the repository, as they're too difficult or not needed.

1. **Empty `divs`:** Elm seems to need adding quite a lot of empty divs, but I'm sure this could be reduced and cleaned up somehow.
2. It could be useful to know how to parse a url segment, for instance `/:uuid`. I've tried figuring out Korban's tutorials (there's a few articles), but the way they're written is a tad complex. Search out other simple examples. His book is a bit too advanced as far as I can tell.
3. See also "virtual room: super high tech room app" which is an interesting demo of using types, but it also uses a lot of fluff and extras that make it look bigger (and harder) than it probably is.

Chapter 5

...

Renaming files, folders, script

Be careful of naming conventions and refactoring the model names. Currently I'm using the chapter names for filenames, and storing each chapter's files in it's own `src/ChapterName/..` folder.

```
// So this ...
```

```
module FileName exposing (main)
// Becomes this ...
module FolderName.FileName exposing (main) element
```

You'll also need to rename the script calling the Elm module in the `index.html` file. Something like this:

```
<script>
// The name of the module
Elm.FolderName.FileName.init({
    node: document.getElementById('selector') // the HTML element
});
<!/script>
```