Table of Contents

Introduction 1.1
Install 1.2
Core Language 1.3
The Elm Architecture 1.4
    Buttons 1.4.1
    Text Fields 1.4.2
    Forms 1.4.3
Types 1.5
    Reading Types 1.5.1
    Type Aliases 1.5.2
    Custom Types 1.5.3
    Pattern Matching 1.5.4
Error Handling 1.6
    Maybe 1.6.1
    Result 1.6.2
Commands and Subscriptions 1.7
    HTTP 1.7.1
    JSON 1.7.2
    Random 1.7.3
    Time 1.7.4
JavaScript Interop 1.8
    Flags 1.8.1
    Ports 1.8.2
    Custom Elements 1.8.3
Web Apps 1.9
    Navigation 1.9.1
    URL Parsing 1.9.2
    Modules 1.9.3
    Structure 1.9.4
Optimization 1.10

Html.lazy	1.10.1
Html.keyed	1.10.2
Minification	1.10.3
Next Steps	1.11
Appendix	1.12
Types as Sets	1.12.1
Types as Bits	1.12.2
Function Types	1.12.3

An Introduction to Elm

Elm is a functional language that compiles to JavaScript. It competes with projects like React as a tool for creating websites and web apps. Elm has a very strong emphasis on simplicity, ease-of-use, and quality tooling.

This guide will:

- Teach you the fundamentals of programming in Elm.
- Show you how to make interactive apps with *The Elm Architecture*.
- Emphasize principles and patterns that generalize to programming in any language.

By the end I hope you will not only be able to create great web apps in Elm, but also understand the core ideas and patterns that make Elm nice to use.

If you are on the fence, I can safely guarantee that if you give Elm a shot and actually make a project in it, you will end up writing better JavaScript and React code. The ideas transfer pretty easily!

A Quick Sample

Of course *I* think Elm is good, so look for yourself.

Here is a simple counter. If you look at the code, it just lets you increment and decrement the counter:

```elm
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox { init = 0, update = update, view = view }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Notice that the `update` and `view` are entirely decoupled. You describe your HTML in a declarative way and Elm takes care of messing with the DOM.

Why a *functional* language?

Forget what you have heard about functional programming. Fancy words, weird ideas, bad tooling. Barf. Elm is about:

- No runtime errors in practice. No `null`. No `undefined` is not a function.
- Friendly error messages that help you add features more quickly.
- Well-architected code that *stays* well-architected as your app grows.
- Automatically enforced semantic versioning for all Elm packages.

No combination of JS libraries can ever give you this, yet it is all free and easy in Elm. Now these nice things are *only* possible because Elm builds upon 40+ years of work on typed functional languages. So Elm is a functional language because the practical benefits are worth the couple hours you'll spend reading this guide.

I have put a huge emphasis on making Elm easy to learn and use, so all I ask is that you give Elm a shot and see what you think. I hope you will be pleasantly surprised!

> **Note:** If you do not want to install yet, you can follow along anyway. Most sections can be done in an online editor!

Install

- Mac — installer
- Windows — installer
- Anywhere — direct download or npm

After installing through any of those routes, you will have the `elm` binary available in your terminal!

> **Troubleshooting:** The fastest way to learn *anything* is to talk with other people in the Elm community. We are friendly and happy to help! So if you get stuck during installation or encounter something weird, visit the Elm Slack and ask about it. In fact, if you run into something confusing at any point while learning or using Elm, come ask us about it. You can save yourself hours. Just do it!

Configure Your Editor

Using Elm is way nicer when you have a code editor to help you out. There are Elm plugins for at least the following editors:

- Atom
- Brackets
- Emacs
- IntelliJ
- Light Table
- Sublime Text
- Vim
- VS Code

If you do not have an editor at all, Sublime Text is a great one to get started with!

You may also want to try out elm-format which makes your code pretty!

Terminal Tools

So we have this `elm` binary now, but what can it do exactly?

`elm repl`

`elm repl` lets us interact with Elm expressions in the terminal.

```
$ elm repl
---- Elm 0.19.0 ----------------------------------------------------------------
Read <https://elm-lang.org/0.19.0/repl> to learn more: exit, help, imports, etc.
--------------------------------------------------------------------------------
> 1 / 2
0.5 : Float
> List.length [1,2,3,4]
4 : Int
> String.reverse "stressed"
"desserts" : String
> :exit
$
```

We will be using `elm repl` in the upcoming "Core Language" section, and you can read more about how it works here.

> **Note:** `elm repl` works by compiling code to JavaScript, so make sure you have Node.js installed. We use that to evaluate code.

`elm reactor`

`elm reactor` helps you build Elm projects without messing with the terminal too much. You just run it at the root of your project, like this:

```
git clone https://github.com/evancz/elm-architecture-tutorial.git
cd elm-architecture-tutorial
elm reactor
```

This starts a server at `http://localhost:8000`. You can navigate to any Elm file and see what it looks like. Try to check out `examples/01-button.elm`.

`elm make`

`elm make` builds Elm projects. It can compile Elm code to HTML or JavaScript. It is the most general way to compile Elm code, so if your project becomes too advanced for `elm reactor`, you will want to start using `elm make` directly.

Say you want to compile `Main.elm` to an HTML file named `main.html`. You would run this command:

```
elm make Main.elm --output=main.html
```

`elm install`

Elm packages all live at `package.elm-lang.org`.

Say you look around and decide you need `elm/http` and `elm/json` to make some HTTP requests. You can get them set up in your project with the following commands:

```
elm install elm/http
elm install elm/json
```

This will add the dependencies into your `elm.json` file, described in more detail here.

Summary

The `elm` binary can do a bunch of stuff. Do not worry about remembering it all. You can always just run `elm --help` or `elm repl --help` to get a bunch of information about any of these commands.

Next we are going to learn the basics of Elm!

Core Language

This section will walk you through Elm's simple core language.

This works best when you follow along, so after installing, run `elm repl` in the terminal. You should see something like this:

```
---- Elm 0.19.0 ----------------------------------------------------------------
Read <https://elm-lang.org/0.19.0/repl> to learn more: exit, help, imports, etc.
--------------------------------------------------------------------------------
>
```

The REPL prints out the type of every result, but **we will leave the type annotations off in this tutorial** for the sake of introducing concepts gradually.

We will cover values, functions, lists, tuples, and records. These building blocks all correspond pretty closely with structures in languages like JavaScript, Python, and Java.

Values

Let's get started with some strings:

```
> "hello"
"hello"

> "hello" ++ "world"
"helloworld"

> "hello" ++ " world"
"hello world"
```

Elm uses the `(++)` operator to put strings together. Notice that both strings are preserved exactly as is when they are put together so when we combine `"hello"` and `"world"` the result has no spaces.

Math looks normal too:

```
> 2 + 3 * 4
14

> (2 + 3) * 4
20
```

Unlike JavaScript, Elm makes a distinction between integers and floating point numbers. Just like Python 3, there is both floating point division `(/)` and integer division `(//)`.

```
> 9 / 2
4.5

> 9 // 2
4
```

Functions

Let's start by writing a function `isNegative` that takes in some number and checks if it is less than zero. The result will be `True` or `False`.

```
> isNegative n = n < 0
<function>

> isNegative 4
False

> isNegative -7
True

> isNegative (-3 * -4)
False
```

Notice that function application looks different than in languages like JavaScript and Python and Java. Instead of wrapping all arguments in parentheses and separating them with commas, we use spaces to apply the function. So `(add(3,4))` becomes `(add 3 4)` which ends up avoiding a bunch of parens and commas as things get bigger. Ultimately, this looks much cleaner once you get used to it! The elm/html package is a good example of how this keeps things feeling light.

You can also define *anonymous functions* like this:

```
> \n -> n < 0
<function>

> (\n -> n < 0) 4
False
```

This anonymous function is the same as `isNegative`, it just is not named! Also, the parentheses in `(\n -> n < 0) 4` are important. After the arrow, Elm is just going to keep reading code as long as it can. The parentheses put bounds on this, indicating where the function body ends. This helps Elm know that `4` is an argument to the function.

> **Note:** The backslash that starts anonymous functions is supposed to look like a lambda `λ` if you squint. This is a possibly ill-conceived wink to the intellectual history that led to languages like Elm.

If Expressions

When you want to have conditional behavior in Elm, you use an if-expression.

```
> if True then "hello" else "world"
"hello"

> if False then "hello" else "world"
"world"
```

The keywords `if` `then` `else` are used to separate the conditional and the two branches so we do not need any parentheses or curly braces.

Elm does not have a notion of “truthiness” so numbers and strings and lists cannot be used as boolean values. If we try it out, Elm will tell us that we need to work with a real boolean value.

Now let's make a function that tells us if a number is over 9000.

```
> over9000 powerLevel = \
|   if powerLevel > 9000 then "It's over 9000!!!" else "meh"
<function>

> over9000 42
"meh"

> over9000 100000
"It's over 9000!!!"
```

Using a backslash in the REPL lets us split things on to multiple lines. We use this in the definition of `over9000` above. Furthermore, it is best practice to always bring the body of a function down a line. It makes things a lot more uniform and easy to read, so you want to do this with all the functions and values you define in normal code.

> **Note:** Make sure that you add a whitespace before the second line of the function. Elm has a "syntactically significant whitespace" meaning that indentation is a part of its syntax.

Lists

Lists are one of the most common data structures in Elm. They hold a sequence of related things, similar to arrays in JavaScript.

Lists can hold many values. Those values must all have the same type. Here are a few examples that use functions from the `List` module:

```
> names = [ "Alice", "Bob", "Chuck" ]
["Alice","Bob","Chuck"]

> List.isEmpty names
False

> List.length names
3

> List.reverse names
["Chuck","Bob","Alice"]

> numbers = [1,4,3,2]
[1,4,3,2]

> List.sort numbers
[1,2,3,4]

> double n = n * 2
<function>

> List.map double numbers
[2,8,6,4]
```

Again, all elements of the list must have the same type.

Tuples

Tuples are another useful data structure. A tuple can hold a fixed number of values, and each value can have any type. A common use is if you need to return more than one value from a function. The following function gets a name and gives a message for the user:

```
> import String

> goodName name = \
|   if String.length name <= 20 then \
|     (True, "name accepted!") \
|   else \
|     (False, "name was too long; please limit it to 20 characters")

> goodName "Tom"
(True, "name accepted!")
```

This can be quite handy, but when things start becoming more complicated, it is often best to use records instead of tuples.

Records

A record is a fixed set of key-value pairs, similar to objects in JavaScript or Python. You will find that they are extremely common and useful in Elm! Let's see some basic examples.

```
> point = { x = 3, y = 4 }
{ x = 3, y = 4 }

> point.x
3

> bill = { name = "Gates", age = 62 }
{ age = 62, name = "Gates" }

> bill.name
"Gates"
```

So we can create records using curly braces and access fields using a dot. Elm also has a version of record access that works like a function. By starting the variable with a dot, you are saying *please access the field with the following name*. This means that `.name` is a function that gets the `name` field of the record.

```
> .name bill
"Gates"

> List.map .name [bill,bill,bill]
["Gates","Gates","Gates"]
```

When it comes to making functions with records, you can do some pattern matching to make things a bit lighter.

```
> under70 {age} = age < 70
<function>

> under70 bill
True

> under70 { species = "Triceratops", age = 68000000 }
False
```

So we can pass any record in as long as it has an `age` field that holds a number.

It is often useful to update the values in a record.

```
> { bill | name = "Nye" }
{ age = 62, name = "Nye" }

> { bill | age = 22 }
{ age = 22, name = "Gates" }
```

It is important to notice that we do not make *destructive* updates. When we update some fields of `bill` we actually create a new record rather than overwriting the existing one. Elm makes this efficient by sharing as much content as possible. If you update one of ten fields, the new record will share the nine unchanged values.

> ## Records vs Objects
>
> Records in Elm are *similar* to objects in JavaScript, but there are some important differences. With records:
>
> - You cannot ask for a field that does not exist.
> - No field will ever be `undefined` or `null`.
> - You cannot create recursive records with a `this` or `self` keyword.
>
> Elm encourages a strict separation of data and logic, and the ability to say `this` is primarily used to break this separation. This is a systemic problem in Object Oriented languages that Elm is purposely avoiding.
>
> Records also support structural typing which means records in Elm can be used in any situation as long as the necessary fields exist. This gives us flexibility without compromising reliability.

The Elm Architecture

The Elm Architecture is a simple pattern for architecting webapps. It is great for modularity, code reuse, and testing. Ultimately, it makes it easy to create complex web apps that stay healthy as you refactor and add features.

This architecture seems to emerge naturally in Elm. Rather than someone “inventing” it, early Elm programmers kept discovering the same basic patterns in their code. Teams have found this particularly nice for onboarding new developers. Code just turns out well-architected. It is kind of spooky.

So The Elm Architecture is *easy* in Elm, but it is useful in any front-end project. In fact, projects like Redux have been inspired by The Elm Architecture, so you may have already seen derivatives of this pattern. Point is, even if you ultimately cannot use Elm at work yet, you will get a lot out of using Elm and internalizing this pattern.

The Basic Pattern

The logic of every Elm program will break up into three cleanly separated parts:

- **Model** — the state of your application
- **Update** — a way to update your state
- **View** — a way to view your state as HTML

This pattern is so reliable that I always start with the following skeleton and fill in details for my particular case.

```elm
import Html exposing (..)


-- MODEL

type alias Model = { ... }


-- UPDATE

type Msg = Reset | ...

update : Msg -> Model -> Model
update msg model =
  case msg of
    Reset -> ...
    ...


-- VIEW

view : Model -> Html Msg
view model =
  ...
```

That is really the essence of The Elm Architecture! We will proceed by filling in this skeleton with increasingly interesting logic.

The Elm Architecture + User Input

Your web app is going to need to deal with user input. This section will get you familiar with The Elm Architecture in the context of things like:

- Buttons
- Text Fields
- Check Boxes
- Radio Buttons
- etc.

We will go through a few examples that build knowledge gradually, so go in order!

Follow Along

In the last section we used `elm repl` to get comfortable with Elm expressions. In this section, we are switching to creating Elm files of our own. You can do that in the online editor, or if you have Elm installed, you can follow these simple instructions to get everything working on your computer!

Buttons

Clone the code or follow along in the online editor.

Our first example is a simple counter that can be incremented or decremented. I find that it can be helpful to see the entire program in one place, so here it is! We will break it down afterwards.

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)


main =
  Browser.sandbox { init = init, update = update, view = view }


-- MODEL

type alias Model = Int

init : Model
init =
  0


-- UPDATE

type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

That's everything!

> **Note:** This section has `type` and `type alias` declarations. You can read all about these in the upcoming section on types. You do not *need* to deeply understand that stuff now, but you are free to jump ahead if it helps.

When writing this program from scratch, I always start by taking a guess at the model. To make a counter, we at least need to keep track of a number that is going up and down. So let's just start with that!

```
type alias Model = Int
```

Now that we have a model, we need to define how it changes over time. I always start my `UPDATE` section by defining a set of messages that we will get from the UI:

```
type Msg = Increment | Decrement
```

I definitely know the user will be able to increment and decrement the counter. The `Msg` type describes these capabilities as *data*. Important! From there, the `update` function just describes what to do when you receive one of these messages.

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

If you get an `Increment` message, you increment the model. If you get a `Decrement` message, you decrement the model. Pretty straight-forward stuff.

Okay, so that's all good, but how do we actually make some HTML and show it on screen? Elm has a library called `elm/html` that gives you full access to HTML5 as normal Elm functions:

```
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

One thing to notice is that our `view` function is producing a `Html Msg` value. This means that it is a chunk of HTML that can produce `Msg` values. And when you look at the definition, you see the `onClick` attributes are set to give out `Increment` and `Decrement` values. These will get fed directly into our `update` function, driving our whole app forward.

Another thing to notice is that `div` and `button` are just normal Elm functions. These functions take (1) a list of attributes and (2) a list of child nodes. It is just HTML with slightly different syntax. Instead of having `<` and `>` everywhere, we have `[` and `]`. We have found that folks who can read HTML have a pretty easy time learning to read this variation. Okay, but why not have it be *exactly* like HTML? **Since we are using normal Elm functions, we have the full power of the Elm programming language to help us build our views!** We can refactor repetitive code out into functions. We can put helpers in modules and import them just like any other code. We can use the same testing frameworks and libraries as any other Elm code. Everything that is nice about programming in Elm is 100% available to help you with your view. No need for a hacked together templating language!

There is also something a bit deeper going on here. **The view code is entirely declarative**. We take in a `Model` and produce some `Html`. That is it. There is no need to mutate the DOM manually. Elm takes care of that behind the scenes. This gives Elm much more leeway to make optimizations and ends up making rendering *faster* overall. So you write less code and the code runs faster. The best kind of abstraction!

This pattern is the essence of The Elm Architecture. Every example we see from now on will be a slight variation on this basic pattern: `Model`, `update`, `view`.

> **Exercise:** One cool thing about The Elm Architecture is that it is super easy to extend as our product requirements change. Say your product manager has come up with this amazing "reset" feature. A new button that will reset the counter to zero.
>
> To add the feature you come back to the `Msg` type and add another possibility: `Reset`. You then move on to the `update` function and describe what happens when you get that message. Finally you add a button in your view.
>
> See if you can implement the "reset" feature!

Text Fields

Clone the code or follow along in the online editor.

We are about to create a simple app that reverses the contents of a text field.

Again this is a pretty short program, so I have included the whole thing here. Skim through to get an idea of how everything fits together. Right after that we will get into the details!

```
import Browser
import Html exposing (Html, Attribute, div, input, text)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)



-- MAIN


main =
  Browser.sandbox { init = init, update = update, view = view }



-- MODEL


type alias Model =
  { content : String
  }


init : Model
init =
  { content = "" }



-- UPDATE


type Msg
  = Change String
```

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }



-- VIEW


view : Model -> Html Msg
view model =
  div []
    [ input [ placeholder "Text to reverse", value model.content, onInput Change ] []
    , div [] [ text (String.reverse model.content) ]
    ]
```

This code is a slight variant of the counter from the previous section. You set up a model. You define some messages. You say how to `update`. You make your `view`. The difference is just in how we filled this skeleton in. Let's walk through that!

As always, you start by guessing at what your `Model` should be. In our case, we know we are going to have to keep track of whatever the user has typed into the text field. We need that information so we know how to render the reversed text.

```
type alias Model =
  { content : String
  }
```

This time I chose to represent the model as a record. (You can read more about records here and here.) For now, the record stores the user input in the `content` field.

> **Note:** You may be wondering, why bother having a record if it only holds one entry? Couldn't you just use the string directly? Sure! But starting with a record makes it easy to add more fields as our app gets more complicated. When the time comes where we want *two* text inputs, we will have to do much less fiddling around.

Okay, so we have our model. Now in this app there is only one kind of message really. The user can change the contents of the text field.

```
type Msg
  = Change String
```

This means our update function just has to handle this one case:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }
```

When we receive new content, we use the record update syntax to update the contents of `content`.

Finally we need to say how to view our application:

```
view : Model -> Html Msg
view model =
  div []
    [ input [ placeholder "Text to reverse", value model.content, onInput Change ] []
    , div [] [ text (String.reverse model.content) ]
    ]
```

We create a `<div>` with two children.

The interesting child is the `<input>` node. In addition to the `placeholder` and `value` attributes, it uses `onInput` to declare what messages should be sent when the user types into this input.

This `onInput` function is kind of interesting. It takes one argument, in this case the `Change` function which was created when we declared the `Msg` type:

```
Change : String -> Msg
```

This function is used to tag whatever is currently in the text field. So let's say the text field currently holds `glad` and the user types `e`. This triggers an `input` event, so we will get the message `Change "glade"` in our `update` function.

So now we have a simple text field that can reverse user input. Neat! Now on to putting a bunch of text fields together into a more traditional form.

Forms

Clone the code or follow along in the online editor.

Here we will make a rudimentary form. It has a field for your name, a field for your password, and a field to verify that password. We will also do some very simple validation (do the two passwords match?) just because it is simple to add.

The code is a bit longer in this case, but I still think it is valuable to look through it before you get into the description of what is going on.

```
import Browser
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)



-- MAIN


main =
  Browser.sandbox { init = init, update = update, view = view }



-- MODEL


type alias Model =
  { name : String
  , password : String
  , passwordAgain : String
  }


init : Model
init =
  Model "" "" ""



-- UPDATE
```

```
type Msg
  = Name String
  | Password String
  | PasswordAgain String


update : Msg -> Model -> Model
update msg model =
  case msg of
    Name name ->
      { model | name = name }

    Password password ->
      { model | password = password }

    PasswordAgain password ->
      { model | passwordAgain = password }



-- VIEW


view : Model -> Html Msg
view model =
  div []
    [ viewInput "text" "Name" model.name Name
    , viewInput "password" "Password" model.password Password
    , viewInput "password" "Re-enter Password" model.passwordAgain PasswordAgain
    , viewValidation model
    ]


viewInput : String -> String -> String -> (String -> msg) -> Html msg
viewInput t p v toMsg =
  input [ type_ t, placeholder p, value v, onInput toMsg ] []


viewValidation : Model -> Html msg
viewValidation model =
  if model.password == model.passwordAgain then
    div [ style "color" "green" ] [ text "OK" ]
  else
    div [ style "color" "red" ] [ text "Passwords do not match!" ]
```

This is pretty similar to our text field example, just with more fields. Let's walk through how it came to be!

As always, you start out by guessing at the `Model`. We know there are going to be three text fields, so let's just go with that:

```
type alias Model =
  { name : String
  , password : String
  , passwordAgain : String
  }
```

Great, seems reasonable. We expect that each of these fields can be changed separately, so our messages should account for each of those scenarios.

```
type Msg
  = Name String
  | Password String
  | PasswordAgain String
```

This means our `update` is pretty mechanical. Just update the relevant field:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Name name ->
      { model | name = name }

    Password password ->
      { model | password = password }

    PasswordAgain password ->
      { model | passwordAgain = password }
```

We get a little bit fancier than normal in our `view` though.

```
view : Model -> Html Msg
view model =
  div []
    [ viewInput "text" "Name" model.name Name
    , viewInput "password" "Password" model.password Password
    , viewInput "password" "Re-enter Password" model.passwordAgain PasswordAgain
    , viewValidation model
    ]
```

We start by creating a `<div>` with four child nodes. But instead of using functions from `elm/html` directly, we call Elm functions to make our code more concise! We start with three calls to `viewInput` :

```
viewInput : String -> String -> String -> (String -> msg) -> Html msg
viewInput t p v toMsg =
  input [ type_ t, placeholder p, value v, onInput toMsg ] []
```

So `viewInput "text" "Name" model.name Name` can create a node like `<input type="text" placeholder="Name" value="Bill">`. That node will also send messages like `Name "Billy"` to `update` on user input.

The fourth entry is more interesting. It is a call to `viewValidation`:

```
viewValidation : Model -> Html msg
viewValidation model =
  if model.password == model.passwordAgain then
    div [ style "color" "green" ] [ text "OK" ]
  else
    div [ style "color" "red" ] [ text "Passwords do not match!" ]
```

This function first compares the two passwords. If they match, you get green text and a positive message. If they do not match, you get red text and a helpful message.

These helper functions begin to show the benefits of having our HTML library be normal Elm code. We *could* put all that code into our `view`, but making helper functions is totally normal in Elm, even in view code. Is this getting hard to understand? Maybe I can break out a helper function!

> **Exercises:** One cool thing about breaking `viewValidation` out is that it is pretty easy to augment. If you are messing with the code as you read through this (as you should be!) you should try to:
>
> - Check that the password is longer than 8 characters.
> - Make sure the password contains upper case, lower case, and numeric characters.
> - Add an additional field for `age` and check that it is a number.
> - Add a `Submit` button. Only show errors *after* it has been pressed.
>
> Be sure to use the helpers in the `String` module if you try any of these! Also, we need to learn more before we start talking to servers, so make sure you read all the way to the HTTP part before trying that. It will be significantly easier with proper guidance!
>
> **Note:** It seems like efforts to make generic validation libraries have not been too successful. I think the problem is that the checks are usually best captured by normal Elm functions. Take some args, give back a `Bool` or `Maybe`. E.g. Why use a library to check if two strings are equal? So as far as we know, the simplest code comes from writing the logic for your particular scenario without any special extras. So definitely give that a shot before deciding you need something more complex!

Types

One of Elm's major benefits is that **users do not see runtime errors in practice**. This is possible because the Elm compiler can analyze your source code very quickly to see how values flow through your program. If a value can ever be used in an invalid way, the compiler tells you about it with a friendly error message. This is called *type inference*. The compiler figures out what *type* of values flow in and out of all your functions.

An Example of Type Inference

The following code defines a `toFullName` function which extracts a person’s full name as a string:

```
toFullName person =
  person.firstName ++ " " ++ person.lastName

fullName =
  toFullName { fistName = "Hermann", lastName = "Hesse" }
```

Like in JavaScript or Python, we just write the code with no extra clutter. Do you see the bug though?

In JavaScript, the equivalent code spits out `"undefined Hesse"` . Not even an error! Hopefully one of your users will tell you about it when they see it in the wild. In contrast, the Elm compiler just looks at the source code and tells you:

```
-- TYPE MISMATCH ---------------------------------------------------------------

The argument to function `toFullName` is causing a mismatch.

6│   toFullName { fistName = "Hermann", lastName = "Hesse" }
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Function `toFullName` is expecting the argument to be:

    { …, firstName : … }

But it is:

    { …, fistName : … }

Hint: I compared the record fields and found some potential typos.

    firstName <-> fistName
```

It sees that `toFullName` is getting the wrong *type* of argument. Like the hint in the error message says, someone accidentally wrote `fist` instead of `first`.

It is great to have an assistant for simple mistakes like this, but it is even more valuable when you have hundreds of files and a bunch of collaborators making changes. No matter how big and complex things get, the Elm compiler checks that *everything* fits together properly just based on the source code.

The better you understand types, the more the compiler feels like a friendly assistant. So let's start learning more!

Reading Types

In the Core Language section of this book, we ran a bunch of code in the REPL. Well, we are going to do it again, but now with an emphasis on the types that are getting spit out. So type `elm repl` in your terminal again. You should see this:

```
---- Elm 0.19.0 ----------------------------------------------------------------
Read <https://elm-lang.org/0.19.0/repl> to learn more: exit, help, imports, etc.
--------------------------------------------------------------------------------
>
```

Primitives and Lists

Let's enter some simple expressions and see what happens:

```
> "hello"
"hello" : String

> not True
False : Bool

> round 3.1415
3 : Int
```

In these three examples, the REPL tells us the resulting value along with what *type* of value it happens to be. The value `"hello"` is a `String`. The value `3` is an `Int`. Nothing too crazy here.

Let's see what happens with lists holding different types of values:

```
> [ "Alice", "Bob" ]
[ "Alice", "Bob" ] : List String

> [ 1.0, 8.6, 42.1 ]
[ 1.0, 8.6, 42.1 ] : List Float

> []
[] : List a
```

In the first case, we have a `List` filled with `String` values. In the second, the `List` is filled with `Float` values. In the third case the list is empty, so we do not actually know what kind of values are in the list. So the type `List a` is saying "I know I have a list, but it could be filled with anything". The lower-case `a` is called a *type variable*, meaning that there are no constraints in our program that pin this down to some specific type. In other words, the type can vary based on how it is used.

Functions

Let's see the type of some functions:

```
> String.length
<function> : String -> Int
```

The function `String.length` has type `String -> Int`. This means it *must* take in a `String` argument, and it will definitely return an integer result. So let's try giving it an argument:

```
> String.length "Supercalifragilisticexpialidocious"
34 : Int
```

So we start with a `String -> Int` function and give it a `String` argument. This results in an `Int`.

What happens when you do not give a `String` though?

```
> String.length [1,2,3]
-- error!

> String.length True
-- error!
```

A `String -> Int` function *must* get a `String` argument!

> **Note:** Functions that take multiple arguments end up having more and more arrows. For example, here is a function that takes two arguments:
>
> ```
> String.repeat : Int -> String -> String
> ```
>
> Giving two arguments like `String.repeat 3 "ha"` will produce `"hahaha"`. It works to think of `->` as a weird way to separate arguments, but I explain the real reasoning here. It is pretty neat!

Type Annotations

So far we have just let Elm figure out the types, but it also lets you write a **type annotation** on the line above a definition if you want. So when you are writing code, you can say things like this:

```elm
half : Float -> Float
half n =
  n / 2

-- half 256 == 128
-- half "3" -- error!

hypotenuse : Float -> Float -> Float
hypotenuse a b =
  sqrt (a^2 + b^2)

-- hypotenuse 3 4  == 5
-- hypotenuse 5 12 == 13

checkPower : Int -> String
checkPower powerLevel =
  if powerLevel > 9000 then "It's over 9000!!!" else "Meh"

-- checkPower 9001 == "It's over 9000!!!"
-- checkPower True -- error!
```

Adding type annotations is not required, but it is definitely recommended! Benefits include:

1. **Error Message Quality** — When you add a type annotation, it tells the compiler what you are *trying* to do. Your implementation may have mistakes, and now the compiler can compare against your stated intent. "You said argument `powerLevel` was an `Int`, but it is getting used as a `String`!"
2. **Documentation** — When you revisit code later (or when a colleague visits it for the first time) it can be really helpful to see exactly what is going in and out of the function without having to read the implementation super carefully.

People can make mistakes in type annotations though, so what happens if the annotation does not match the implementation? The compiler figures out all the types on its own, and it checks that your annotation matches the real answer. In other words, the compiler will always verify that all the annotations you add are correct. So you get better error messages *and* documentation always stays up to date!

Type Variables

As you look through the functions in `elm/core`, you will see some type signatures with lower-case letters in them. We can check some of them out in `elm repl`:

```
> List.length
<function> : List a -> Int
```

Notice that lower-case `a` in the type? That is called a **type variable**. It can vary depending on how `List.length` is used:

```
> List.length [1,1,2,3,5,8]
6 : Int

> List.length [ "a", "b", "c" ]
3 : Int

> List.length [ True, False ]
2 : Int
```

We just want the length, so it does not matter what is in the list. So the type variable `a` is saying that we can match any type. Let’s look at another common example:

```
> List.reverse
<function> : List a -> List a

> List.reverse [ "a", "b", "c" ]
["c","b","a"] : List String

> List.reverse [ True, False ]
[False,True] : List Bool
```

Again, the type variable `a` can vary depending on how `List.reverse` is used. But in this case, we have an `a` in the argument and in the result. This means that if you give a `List Int` you must get a `List Int` as well. Once we decide what `a` is, that’s what it is everywhere.

> **Note:** Type variables must start with a lower-case letter, but they can be full words. We could write the type of `List.length` as `List value -> Int` and we could write the type of `List.reverse` as `List element -> List element`. It is fine as long as they start with a lower-case letter. Type variables `a` and `b` are used by convention in many places, but some type annotations benefit from more specific names.

Constrained Type Variables

There are a few “constrained” type variables. The most common example is probably the `number` type. The `negate` function uses it:

```
negate : number -> number
```

Normally type variables can get filled in with *anything*, but `number` can only be filled in by `Int` and `Float` values. It constrains the possibilities.

The full list of constrained type variables is:

- `number` permits `Int` and `Float`
- `appendable` permits `String` and `List a`
- `comparable` permits `Int`, `Float`, `Char`, `String`, and lists/tuples of `comparable` values
- `compappend` permits `String` and `List comparable`

These constrained type variables exist to make operators like `(+)` and `(<)` a bit more flexible.

Type Aliases

Elm allows you to create a **type alias**. An alias is just a shorter name for some other type. It looks like this:

```
type alias User =
  { name : String
  , bio : String
  }
```

So rather than having to type out this record type all the time, we can just say `User` instead. For example, you can shorten type annotations like this:

```
hasDecentBio : User -> Bool
hasDecentBio user =
  String.length user.bio > 80
```

That would be `{ name : String, bio : String } -> Bool` without the type alias. **The main point of type aliases is to help us write shorter and clearer type annotations.** This becomes more important as your application grows. Say we have a `updateBio` function:

```
updateBio : String -> User -> User
updateBio bio user =
  { user | bio = bio }
```

First, think about the type signature without a type alias! Now, imagine that as our application grows we add more fields to represent a user. We could add 10 or 100 fields to the `User` type alias, and we do not need any changes to our `updateBio` function. Nice!

Record Constructors

When you create a type alias specifically for a record, it also generates a **record constructor**. So if we define a `User` type alias in `elm repl` we could start building records like this:

```
> type alias User = { name : String, bio : String }

> User "Tom" "Friendly Carpenter"
{ name = "Tom", bio = "Friendly Carpenter" }
```

The arguments are in the order they appear in the type alias declaration. This can be pretty handy.

And again, this is only for records. Making type aliases for non-record types will not result in a constructor.

> **Note:** Custom types used to be referred to as “union types” in Elm. Names from other communities include tagged unions and ADTs.

Custom Types

So far we have seen a bunch of types like `Bool`, `Int`, and `String`. But how do we define our own?

Say we are making a chat room. Everyone needs a name, but maybe some users do not have a permanent account. They just give a name each time they show up.

We can describe this situation by defining a `UserStatus` type, listing all the possible variations:

```
type UserStatus = Regular | Visitor
```

The `UserStatus` type has two **variants**. Someone can be a `Regular` or a `Visitor`. So we could represent a user as a record like this:

```
type UserStatus
  = Regular
  | Visitor

type alias User =
  { status : UserStatus
  , name : String
  }

thomas = { status = Regular, name = "Thomas" }
kate95 = { status = Visitor, name = "kate95" }
```

So now we can track if someone is a `Regular` with an account or a `Visitor` who is just passing through. It is not too tough, but we can make it simpler!

Rather than creating a custom type and a type alias, we can represent all this with just a single custom type. The `Regular` and `Visitor` variants each have an associated data. In our case, the associated data is a `String` value:

```
type User
  = Regular String
  | Visitor String

thomas = Regular "Thomas"
kate95 = Visitor "kate95"
```

The data is attached directly to the variant, so there is no need for the record anymore.

Another benefit of this approach is that each variant can have different associated data. Say that `Regular` users gave their age when they signed up. There is no nice way to capture that with records, but when you define your own custom type it is no problem. We add some associated data to the `Regular` variant:

```
type User
  = Regular String Int
  | Visitor String

thomas = Regular "Thomas" 44
kate95 = Visitor "kate95"
```

The different variants of a type can diverge quite dramatically. For example, maybe we add location for `Regular` users so we can suggest regional chat rooms. Add more associated data! Or maybe we want to have anonymous users. Add a third variant called `Anonymous`. Maybe we end up with:

```
type User
  = Regular String Int Location
  | Visitor String
  | Anonymous
```

No problem! Let’s see some other examples now.

Messages

In the architecture section, we saw a couple of examples of defining a `Msg` type. This sort of type is extremely common in Elm. In our chat room, we might define a `Msg` type like this:

```
type Msg
  = PressedEnter
  | ChangedDraft String
  | ReceivedMessage { user : User, message : String }
  | ClickedExit
```

We have four variants. Some variants have no associated data, others have a bunch. Notice that `ReceivedMessage` actually has a record as associated data. That is totally fine. Any type can be associated data! This allows you to describe interactions in your application very precisely.

Modeling

Custom types become extremely powerful when you start modeling situations very precisely. For example, if you are waiting for some data to load, you might want to model it with a custom type like this:

```elm
type Profile
  = Failure
  | Loading
  | Success { name : String, description : String }
```

So you can start in the `Loading` state and then transition to `Failure` or `Success` depending on what happens. This makes it really simple to write a `view` function that always shows something reasonable when data is loading.

Now we know how to create custom types, the next section will show how to use them!

> **Note: Custom types are the most important feature in Elm.** They have a lot of depth, especially once you get in the habit of trying to model scenarios more precisely. I tried to share some of this depth in Types as Sets and Types as Bits in the appendix. I hope you find them helpful!

Pattern Matching

On the previous page, we learned how to create custom types with the `type` keyword. Our primary example was a `User` in a chat room:

```
type User
  = Regular String Int
  | Visitor String
```

Regulars have a name and age, whereas visitors only have a name. So we have our custom type, but how do we actually use it?

`case`

Say we want a `toName` function that decides on a name to show for each `User`. We need to use a `case` expression:

```
toName : User -> String
toName user =
  case user of
    Regular name age ->
      name

    Visitor name ->
      name

-- toName (Regular "Thomas" 44) == "Thomas"
-- toName (Visitor "kate95")    == "kate95"
```

The `case` expression allows us to branch based on which variant we happen to see, so whether we see Thomas or Kate, we always know how to show their name.

And if we try invalid arguments like `toName (Visitar "kate95")` or `toName Anonymous`, the compiler tells us about it immediately. This means many simple mistakes can be fixed in seconds, rather than making it to users and costing a lot more time overall.

Wild Cards

The `toName` function we just defined works great, but notice that the `age` is not used in the implementation? When some of the associated data is unused, it is common to use a “wild card” instead of giving it a name:

```
toName : User -> String
toName user =
  case user of
    Regular name _ ->
      name

    Visitor name ->
      name
```

The `_` acknowledges the data there, but also saying explicitly that nobody is using it.

Error Handling

One of the guarantees of Elm is that you will not see runtime errors in practice. This is partly because **Elm treats errors as data**. Rather than crashing, we model the possibility of failure explicitly with custom types. For example, say you want to turn user input into an age. You might create a custom type like this:

```
type MaybeAge
  = Age Int
  | InvalidInput

toAge : String -> MaybeAge
toAge userInput =
  ...

-- toAge "24" == Age 24
-- toAge "99" == Age 99
-- toAge "ZZ" == InvalidInput
```

Instead of crashing on bad input, we say explicitly that the result may be an `Age 24` or an `InvalidInput`. No matter what input we get, we always produce one of these two variants. From there, we use pattern matching which will ensure that both possibilities are accounted for. No crashing!

This kind of thing comes up all the time! For example, maybe you want to turn a bunch of user input into a `Post` to share with others. But what happens if they forget to add a title? Or there is no content in the post? We could model all these problems explicitly:

```
type MaybePost
  = Post { title : String, content : String }
  | NoTitle
  | NoContent

toPost : String -> String -> MaybePost
toPost title content =
  ...

-- toPost "hi" "sup?" == Post { title = "hi", content = "sup?" }
-- toPost ""   ""     == NoTitle
-- toPost "hi" ""     == NoContent
```

Instead of just saying that the input is invalid, we are describing each of the ways things might have gone wrong. If we have a `viewPreview : MaybePost -> Html msg` function to preview valid posts, now we can give more specific error messages in the preview area when something goes wrong!

These kinds of situations are extremely common. It is often valuable to create a custom type for your exact situation, but in some of the simpler cases, you can use an off-the-shelf type instead. So the rest of this chapter explores the `Maybe` and `Result` types, showing how they can help you treat errors as data!

Maybe

As you work more with Elm, you will start seeing the `Maybe` type quite frequently. It is defined like this:

```
type Maybe a
  = Just a
  | Nothing

-- Just 3.14 : Maybe Float
-- Just "hi" : Maybe String
-- Just True : Maybe Bool
-- Nothing   : Maybe a
```

This is a type with two variants. You either have `Nothing` or you have `Just` a value. The type variable makes it possible to have a `Maybe Float` and `Maybe String` depending on the particular value.

This can be handy in two main scenarios: partial functions and optional fields.

Partial Functions

Sometimes you want a function that gives an answer for some inputs, but not others. Many people run into this with `String.toFloat` when trying to convert user input into numbers. Open up `elm repl` to see it in action:

```
> String.toFloat
<function> : String -> Maybe Float

> String.toFloat "3.1415"
Just 3.1415 : Maybe Float

> String.toFloat "abc"
Nothing : Maybe Float
```

Not all strings make sense as numbers, so this function models that explicitly. Can a string be turned into a float? Maybe! From there we can pattern match on the resulting data and continue as appropriate.

> **Exercise:** I wrote a little program here that converts from Celsius to Fahrenheit. Try refactoring the `view` code in different ways. Can you put a red border around invalid input? Can you add more conversions? Fahrenheit to Celsius? Inches to Meters?

Optional Fields

Another place you commonly see `Maybe` values is in records with optional fields.

For example, say we are running a social networking website. Connecting people, friendship, etc. You know the spiel. The Onion outlined our real goals best back in 2011: mine as much data as possible for the CIA. And if we want *all* the data, we need to ease people into it. Let them add it later. Add features that encourage them to share more and more information over time.

So let's start with a simple model of a user. They must have a name, but we are going to make the age optional.

```
type alias User =
  { name : String
  , age : Maybe Int
  }
```

Now say Sue creates an account, but decides not to provide her birthday:

```
sue : User
sue =
  { name = "Sue", age = Nothing }
```

Sue's friends cannot wish her a happy birthday though. I wonder if they *really* care about her... Later Tom creates a profile and *does* give his age:

```
tom : User
tom =
  { name = "Tom", age = Just 24 }
```

Great, that will be nice on his birthday. But more importantly, Tom is part of a valuable demographic! The advertisers will be pleased.

Alright, so now that we have some users, how can we market alcohol to them without breaking any laws? People would probably be mad if we market to people under 21, so let's check for that:

```
canBuyAlcohol : User -> Bool
canBuyAlcohol user =
  case user.age of
    Nothing ->
      False

    Just age ->
      age >= 21
```

Notice that the `Maybe` type forces us to pattern match on the users age. It is actually impossible to write code where you forget that users may not have an age. Elm makes sure of it! Now we can advertise alcohol confident that we are not influencing minors directly! Only their older peers.

Avoiding Overuse

This `Maybe` type is quite useful, but there are limits. Beginners are particularly prone to getting excited about `Maybe` and using it everywhere, even though a custom type would be more appropriate.

For example, say we have an exercise app where we compete against our friends. You start with a list of your friend's names, but you can load more fitness information about them later. You might be tempted to model it like this:

```
type alias Friend =
  { name : String
  , age : Maybe Int
  , height : Maybe Float
  , weight : Maybe Float
  }
```

All the information is there, but you are not really modeling the way your particular application works. It would be much more precise to model it like this instead:

```
type Friend
  = Less String
  | More String Info

type alias Info =
  { age : Int
  , height : Float
  , weight : Float
  }
```

This new model is capturing much more about your application. There are only two real situations. Either you have just the name, or you have the name and a bunch of information. In your view code, you just think about whether you are showing a `Less` or `More` view of the friend. You do not have to answer questions like “what if I have an `age` but not a `weight`?” That is not possible with our more precise type!

Point is, if you find yourself using `Maybe` everywhere, it is worth examining your `type` and `type alias` definitions to see if you can find a more precise representation. This often leads to a lot of nice refactors in your update and view code!

> ## Aside: Connection to `null` references
>
> The inventor of `null` references, Tony Hoare, described them like this:
>
> > I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> That design makes failure **implicit**. Any time you think you have a `String` you just might have a `null` instead. Should you check? Did the person giving you the value check? Maybe it will be fine? Maybe it will crash your server? I guess we will find out later!
>
> Elm avoids these problems by not having `null` references at all. We instead use custom types like `Maybe` to make failure **explicit**. This way there are never any surprises. A `String` is always a `String`, and when you see a `Maybe String`, the compiler will ensure that both variants are accounted for. This way you get the same flexibility, but without the surprise crashes.

Result

The `Maybe` type can help with simple functions that may fail, but it does not tell you *why* it failed. Imagine if a compiler just said `Nothing` if anything was wrong with your program. Good luck figuring out what went wrong!

This is where the `Result` type becomes helpful. It is defined like this:

```
type Result error value
  = Ok value
  | Err error
```

The point of this type is to give additional information when things go wrong. It is really helpful for error reporting and error recovery!

Error Reporting

Perhaps we have a website where people input their age. We could check that the age is reasonable with a function like this:

```
isReasonableAge : String -> Result String Int
isReasonableAge input =
  case String.toInt input of
    Nothing ->
      Err "That is not a number!"

    Just age ->
      if age < 0 then
        Err "Please try again after you are born."

      else if age > 135 then
        Err "Are you some kind of turtle?"

      else
        Ok age

-- isReasonableAge "abc" == Err ...
-- isReasonableAge "-13" == Err ...
-- isReasonableAge "24"  == Ok 24
-- isReasonableAge "150" == Err ...
```

Not only can we check the age, but we can also show people error messages depending on the particulars of their input. This kind of feedback is much better than `Nothing`!

Error Recovery

The `Result` type can also help you recover from errors. One place you see this is when making HTTP requests. Say we want to show the full text of *Anna Karenina* by Leo Tolstoy. Our HTTP request results in a `Result Error String` to capture the fact that the request may succeed with the full text, or it may fail in a bunch of different ways:

```
type Error
  = BadUrl String
  | Timeout
  | NetworkError
  | BadStatus Int
  | BadBody String

-- Ok "All happy ..." : Result Error String
-- Err Timeout        : Result Error String
-- Err NetworkError   : Result Error String
```

From there we can show nicer error messages as we discussed before, but we can also try to recover from the failure! If we see a `Timeout` it may work to wait a little while and try again. Whereas if we see a `BadStatus 404` then there is no point in trying again.

The next chapter shows how to actually make HTTP requests, so we will run into the `Result` and `Error` types again very soon!

Commands and Subscriptions

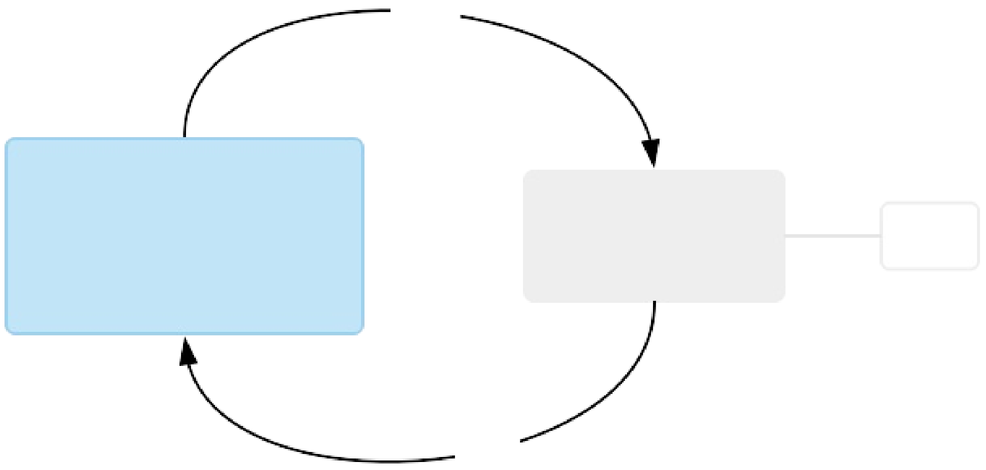
Earlier in this book we saw The Elm Architecture handle mouse and keyboard interactions, but what about talking to servers? Generating random numbers?

To answer these questions, it helps to learn more about how The Elm Architecture works behind the scenes. This will explain why things work a bit differently than in languages like JavaScript, Python, etc.

`sandbox`

I have not made a big deal about it, but so far all of our programs were created with `Browser.sandbox`. We gave an initial `Model` and describe how to `update` and `view` it.

You can think of `Browser.sandbox` as setting up a system like this:



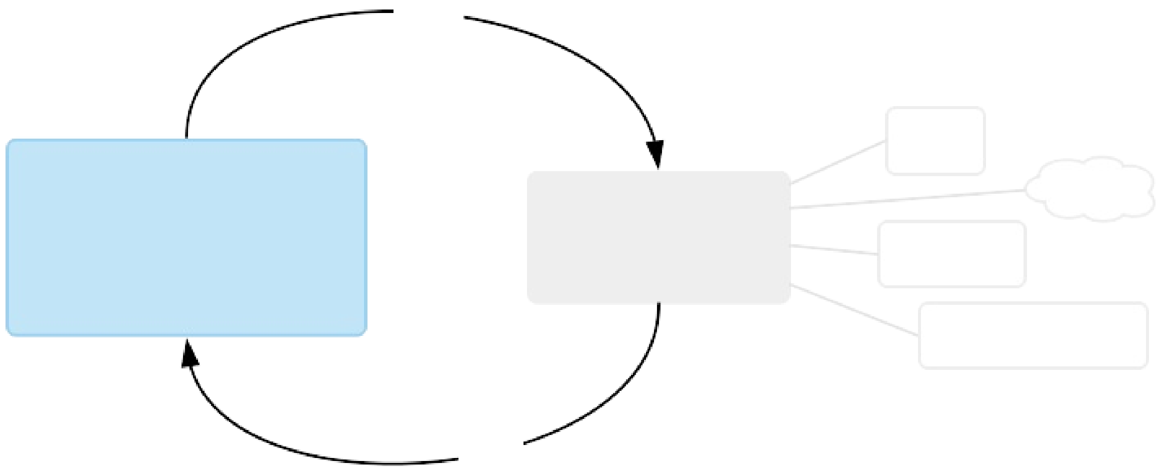
We get to stay in the world of Elm, writing functions and transforming data. This hooks up to Elm's **runtime system**. The runtime system figures out how to render `Html` efficiently. Did anything change? What is the minimal DOM modification needed? It also figures out when someone clicks a button or types into a text field. It turns that into a `Msg` and feeds it into your Elm code.

By cleanly separating out all the DOM manipulation, it becomes possible to use extremely aggressive optimizations. So Elm's runtime system is a big part of why Elm is one of the fastest options available.

`element`

In the next few examples, we are going to use `Browser.element` to create programs. This will introduce the ideas of **commands** and **subscriptions** which allow us to interact with the outside world.

You can think of `Browser.element` as setting up a system like this:



In addition to producing `Html` values, our programs will also send `Cmd` and `Sub` values to the runtime system. In this world, our programs can **command** the runtime system to make an HTTP request or to generate a random number. They can also **subscribe** to the current time.

I think commands and subscriptions make more sense when you start seeing examples, so let's do that!

> **Note 1:** Some readers may be worrying about asset size. “A runtime system? That sounds big!” It is not! In fact, Elm assets are exceptionally small when compared to popular alternatives.
>
> **Note 2:** We are going to use packages from `package.elm-lang.org` in the upcoming examples. We have already been working with a couple:
>
> - `elm/core`
> - `elm/html`
>
> But now we will start getting into some fancier ones:
>
> - `elm/http`
> - `elm/json`
> - `elm/random`
> - `elm/time`
>
> There are tons of other packages on `package.elm-lang.org` though! So when you are making your own Elm programs locally, it will probably involve running some commands like this in the terminal:
>
> ```
> elm init
> elm install elm/http
> elm install elm/random
> ```
>
> That would set up an `elm.json` file with `elm/http` and `elm/random` as dependencies.
>
> I will be mentioning the packages we are using in the following examples, so I hope this gives some context on what that is all about!

HTTP

Clone the code or follow along in the online editor.

It is often helpful to grab information from elsewhere on the internet.

For example, say we want to load the full text of *Public Opinion* by Walter Lippmann. Published in 1922, this book provides a historical perspective on the rise of mass media and its implications for democracy. For our purposes here, we will focus on how to use the `elm/http` package to get this book into our app!

Let's start by just looking at all the code. There are some new things, but do not worry. We will go through it all!

```
import Browser
import Html exposing (Html, text, pre)
import Http



-- MAIN


main =
  Browser.element
    { init = init
    , update = update
    , subscriptions = subscriptions
    , view = view
    }



-- MODEL


type Model
  = Failure
  | Loading
  | Success String


init : () -> (Model, Cmd Msg)
init _ =
  ( Loading
  , Http.get
      { url = "https://elm-lang.org/assets/public-opinion.txt"
      , expect = Http.expectString GotText
      }
  )



-- UPDATE


type Msg
  = GotText (Result Http.Error String)


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    GotText result ->
      case result of
        Ok fullText ->
          (Success fullText, Cmd.none)

        Err _ ->
          (Failure, Cmd.none)



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none



-- VIEW


view : Model -> Html Msg
view model =
  case model of
    Failure ->
      text "I was unable to load your book."

    Loading ->
      text "Loading..."

    Success fullText ->
      pre [] [ text fullText ]
```

Some parts of this should be familiar from previous examples of The Elm Architecture. We still have a `Model` of our application. We still have an `update` that reacts to messages. We still have a `view` function that shows everything on screen.

The new parts extend the core pattern we saw before with some changes in `init` and `update`, and the addition of `subscription`.

`init`

The `init` function describes how to initialize our program:

```
init : () -> (Model, Cmd Msg)
init _ =
  ( Loading
  , Http.get
      { url = "http://www.gutenberg.org/cache/epub/6456/pg6456.txt"
      , expect = Http.expectString GotText
      }
  )
```

Like always, we have to produce the initial `Model`, but now we are also producing some **command** of what we want to do immediately. That command will eventually produce a `Msg` that gets fed into the `update` function.

Our book website starts in the `Loading` state, and we want to GET the full text of our book. When making a GET request with `Http.get`, we specify the `url` of the data we want to fetch, and we specify what we `expect` that data to be. So in our case, the `url` is pointing at some data on the Project Gutenberg website, and we `expect` it to be a big `String` we can show on screen.

The `Http.expectString GotText` line is saying a bit more than that we `expect` a `String` though. It is also saying that when we get a response, it should be turned into a `GotText` message:

```
type Msg
  = GotText (Result Http.Error String)

-- GotText (Ok "The Project Gutenberg EBook of ...")
-- GotText (Err Http.NetworkError)
-- GotText (Err (Http.BadStatus 404))
```

Notice that we are using the `Result` type from a couple sections back. This allows us to fully account for the possible failures in our `update` function. Speaking of `update` functions...

> **Note:** If you are wondering why `init` is a function (and why we are ignoring the argument) we will talk about it in the upcoming chapter on JavaScript interop! (Preview: the argument lets us get information from JS on initialization.)

`update`

Our `update` function is returning a bit more information as well:

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    GotText result ->
      case result of
        Ok fullText ->
          (Success fullText, Cmd.none)

        Err _ ->
          (Failure, Cmd.none)
```

Looking at the type signature, we see that we are not just returning an updated model. We are *also* producing a **command** of what we want Elm to do.

Moving on to the implementation, we pattern match on messages like normal. When a `GotText` message comes in, we inspect the `Result` of our HTTP request and update our model depending on whether it was a success or failure. The new part is that we also provide a command.

So in the case that we got the full text successfully, we say `Cmd.none` to indicate that there is no more work to do. We already got the full text!

And in the case that there was some error, we also say `Cmd.none` and just give up. The text of the book did not load. If we wanted to get fancier, we could pattern match on the `Http.Error` and retry the request if we got a timeout or something.

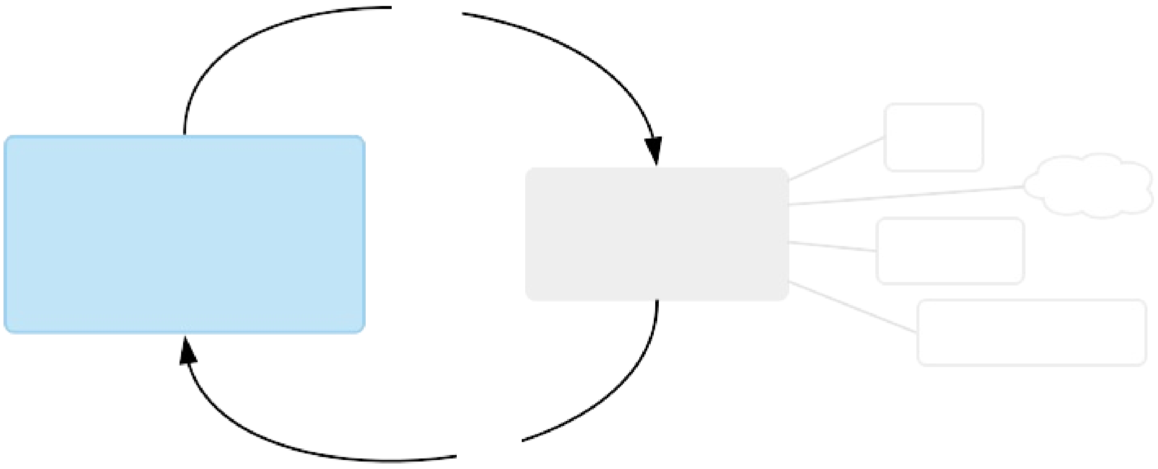
The point here is that however we decide to update our model, we are also free to issue new commands. I need more data! I want a random number! Etc.

`subscription`

The other new thing in this program is the `subscription` function. It lets you look at the `Model` and decide if you want to subscribe to certain information. In our example, we say `Sub.none` to indicate that we do not need to subscribe to anything, but we will soon see an example of a clock where we want to subscribe to the current time!

Summary

When we create a program with `Browser.element`, we set up a system like this:



We get the ability to issue **commands** from `init` and `update`. This allows us to do things like make HTTP requests whenever we want. We also get the ability to **subscribe** to interesting information. (We will see an example of subscriptions later!)

JSON

Clone the code or follow along in the online editor.

We just saw an example that uses HTTP to get the content of a book. That is great, but a ton of servers return data in a special format called JavaScript Object Notation, or JSON for short.

So our next example shows how to fetch some JSON data, allowing us to press a button to show random cat GIFs.

```
import Browser
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Http
import Json.Decode exposing (Decoder, field, string)



-- MAIN


main =
  Browser.element
    { init = init
    , update = update
    , subscriptions = subscriptions
    , view = view
    }



-- MODEL


type Model
  = Failure
  | Loading
  | Success String


init : () -> (Model, Cmd Msg)
init _ =
  (Loading, getRandomCatGif)



-- UPDATE


type Msg
  = MorePlease
  | GotGif (Result Http.Error String)


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    MorePlease ->
      (Loading, getRandomCatGif)

    GotGif result ->
      case result of
        Ok url ->
          (Success url, Cmd.none)

        Err _ ->
          (Failure, Cmd.none)



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none



-- VIEW


view : Model -> Html Msg
view model =
  div []
    [ h2 [] [ text "Random Cats" ]
    , viewGif model
    ]


viewGif : Model -> Html Msg
viewGif model =
  case model of
    Failure ->
      div []
```

```
        [ text "I could not load a random cat for some reason. "
        , button [ onClick MorePlease ] [ text "Try Again!" ]
        ]

    Loading ->
      text "Loading..."

    Success url ->
      div []
        [ button [ onClick MorePlease, style "display" "block" ] [ text "More Please!"
]
        , img [ src url ] []
        ]



-- HTTP


getRandomCatGif : Cmd Msg
getRandomCatGif =
  Http.get
    { url = "https://api.giphy.com/v1/gifs/random?api_key=dc6zaTOxFJmzC&tag=cat"
    , expect = Http.expectJson GotGif gifDecoder
    }


gifDecoder : Decoder String
gifDecoder =
  field "data" (field "image_url" string)
```

This example is pretty similar to the last one:

- `init` starts us off in the `Loading` state, with a command to get a random cat GIF.
- `update` handles the `GotGif` message for whenever a new GIF is available. Whatever happens there, we do not have any additional commands. It also handles the `MorePlease` message when someone presses the button, issuing a command to get more random cats.
- `view` shows you the cats!

The main difference is in the `getRandomCatGif` definition. Instead of using `Http.expectString`, we have switched to `Http.expectJson`. What is the deal with that?

JSON

When you ask `api.giphy.com` for a random cat GIF, their server produces a big string of JSON like this:

```
{
  "data": {
    "type": "gif",
    "id": "l2JhxfHWMBWuDMIpi",
    "title": "cat love GIF by The Secret Life Of Pets",
    "image_url": "https://media1.giphy.com/media/l2JhxfHWMBWuDMIpi/giphy.gif",
    "caption": "",
    ...
  },
  "meta": {
    "status": 200,
    "msg": "OK",
    "response_id": "5b105e44316d3571456c18b3"
  }
}
```

We have no guarantees about any of the information here. The server can change the names of fields, and the fields may have different types in different situations. It is a wild world!

In JavaScript, the approach is to just turn JSON into JavaScript objects and hope nothing goes wrong. But if there is some typo or unexpected data, you get a runtime exception somewhere in your code. Was the code wrong? Was the data wrong? It is time to start digging around to find out!

In Elm, we validate the JSON before it comes into our program. So if the data has an unexpected structure, we learn about it immediately. There is no way for bad data to sneak through and cause a runtime exception three files over. This is accomplished with JSON decoders.

JSON Decoders

Say we have some JSON:

```
{
    "name": "Tom",
    "age": 42
}
```

We need to run it through a `Decoder` to access specific information. So if we wanted to get the `"age"`, we would run the JSON through a `Decoder Int` that describes exactly how to access that information:

If all goes well, we get an `Int` on the other side! And if we wanted the `"name"` we would run the JSON through a `Decoder String` that describes exactly how to access it:

If all goes well, we get a `String` on the other side!

How do we create decoders like this though?

Building Blocks

The `elm/json` package gives us the `Json.Decode` module. It is filled with tiny decoders that we can snap together.

So to get `"age"` from `{ "name": "Tom", "age": 42 }` we would create a decoder like this:

```
import Json.Decode exposing (Decoder, field, int)

ageDecoder : Decoder Int
ageDecoder =
  field "age" int

 -- int : Decoder Int
 -- field : String -> Decoder a -> Decoder a
```

The `field` function takes two arguments:

1. `String` — a field name. So we are demanding an object with an `"age"` field.
2. `Decoder a` — a decoder to try next. So if the `"age"` field exists, we will try this decoder on the value there.

So putting it together, `field "age" int` is asking for an `"age"` field, and if it exists, it runs the `Decoder Int` to try to extract an integer.

We do pretty much exactly the same thing to extract the `"name"` field:

```
import Json.Decode exposing (Decoder, field, string)

nameDecoder : Decoder String
nameDecoder =
  field "name" string

-- string : Decoder String
```

In this case we demand an object with a `"name"` field, and if it exists, we want the value there to be a `String`.

Nesting Decoders

Remember the `api.giphy.com` data?

```
{
  "data": {
    "type": "gif",
    "id": "l2JhxfHWMBWuDMIpi",
    "title": "cat love GIF by The Secret Life Of Pets",
    "image_url": "https://media1.giphy.com/media/l2JhxfHWMBWuDMIpi/giphy.gif",
    "caption": "",
    ...
  },
  "meta": {
    "status": 200,
    "msg": "OK",
    "response_id": "5b105e44316d3571456c18b3"
  }
}
```

We wanted to access `response.data.image_url` to show a random GIF. Well, we have the tools now!

```
import Json.Decode exposing (Decoder, field, string)

gifDecoder : Decoder String
gifDecoder =
  field "data" (field "image_url" string)
```

This is the exact `gifDecoder` definition we used in our example program above! Is there a `"data"` field? Does that value have an `"image_url"` field? Is the value there a string? All our expectations are written out explicitly, allowing us to safely extract Elm values from

JSON.

Combining Decoders

That is all we needed for our HTTP example, but decoders can do more! For example, what if we want *two* fields? We snap decoders together with `map2`:

```
map2 : (a -> b -> value) -> Decoder a -> Decoder b -> Decoder value
```

This function takes in two decoders. It tries them both and combines their results. So now we can put together two different decoders:

```
import Json.Decode exposing (Decoder, map2, field, string, int)

type alias Person =
  { name : String
  , age : Int
  }

personDecoder : Decoder Person
personDecoder =
  map2 Person
      (field "name" string)
      (field "age" int)
```

So if we used `personDecoder` on `{ "name": "Tom", "age": 42 }` we would get out an Elm value like `Person "Tom" 42`.

If we really wanted to get into the spirit of decoders, we would define `personDecoder` as `map2 Person nameDecoder ageDecoder` using our previous definitions. You always want to be building your decoders up from smaller building blocks!

Next Steps

There are a bunch of important functions in `Json.Decode` that we did not cover here:

- `bool` : `Decoder Bool`
- `list` : `Decoder a -> Decoder (List a)`
- `dict` : `Decoder a -> Decoder (Dict String a)`
- `oneOf` : `List (Decoder a) -> Decoder a`

So there are ways to extract all sorts of data structures. The `oneOf` function is particularly helpful for messy JSON. (e.g. sometimes you get an `Int` and other times you get a `String` containing digits. So annoying!)

There are also `map3`, `map4`, and others for handling objects with more than two fields. But as you start working with larger JSON objects, it is worth checking out `NoRedInk/elm-json-decode-pipeline`. The types there are a bit fancier, but some folks find them much easier to read and work with.

> **Fun Fact:** I have heard a bunch of stories of folks finding bugs in their *server* code as they switched from JS to Elm. The decoders people write end up working as a validation phase, catching weird stuff in JSON values. So when NoRedInk switched from React to Elm, it revealed a couple bugs in their Ruby code!

Random

Clone the code or follow along in the online editor.

So far we have only seen commands to make HTTP requests, but we can command other things as well, like generating random values! So we are going to make an app that rolls dice, producing a random number between 1 and 6.

We need the `elm/random` package for this. The `Random` module in particular. Let's start by just looking at all the code:

```
import Browser
import Html exposing (..)
import Html.Events exposing (..)
import Random



-- MAIN


main =
  Browser.element
    { init = init
    , update = update
    , subscriptions = subscriptions
    , view = view
    }



-- MODEL


type alias Model =
  { dieFace : Int
  }


init : () -> (Model, Cmd Msg)
init _ =
  ( Model 1
  , Cmd.none
  )
```

```
-- UPDATE


type Msg
  = Roll
  | NewFace Int


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Roll ->
      ( model
      , Random.generate NewFace (Random.int 1 6)
      )

    NewFace newFace ->
      ( Model newFace
      , Cmd.none
      )



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none



-- VIEW


view : Model -> Html Msg
view model =
  div []
    [ h1 [] [ text (String.fromInt model.dieFace) ]
    , button [ onClick Roll ] [ text "Roll" ]
    ]
```

The new thing here is command issued in the `update` function:

```
Random.generate NewFace (Random.int 1 6)
```

Generating random values works a bit different than in languages like JavaScript, Python, Java, etc. So let's see how it works in Elm!

Random Generators

The core idea is that we have random `Generator` that describes *how* to generate a random value. For example:

```
import Random

probability : Random.Generator Float
probability =
  Random.float 0 1

roll : Random.Generator Int
roll =
  Random.int 1 6

usuallyTrue : Random.Generator Bool
usuallyTrue =
  Random.weighted (80, True) [ (20, False) ]
```

So here we have three random generators. The `roll` generator is saying it will produce an `Int`, and more specifically, it will produce an integer between `1` and `6` inclusive. Likewise, the `usuallyTrue` generator is saying it will produce a `Bool`, and more specifically, it will be true 80% of the time.

The point is that we are not actually generating the values yet. We are just describing *how* to generate them. From there you use the `Random.generate` to turn it into a command:

```
generate : (a -> msg) -> Generator a -> Cmd msg
```

When the command is performed, the `Generator` produces some value, and then that gets turned into a message for your `update` function. So in our example, the `Generator` produces a value between 1 and 6, and then it gets turned into a message like `NewFace 1` or `NewFace 4`. That is all we need to know to get our random dice rolls, but generators can do quite a bit more!

Combining Generators

Once we have some simple generators like `probability` and `usuallyTrue`, we can start snapping them together with functions like `map3`. Imagine we want to make a simple slot machine. We could create a generator like this:

```
import Random

type Symbol = Cherry | Seven | Bar | Grapes

symbol : Random.Generator Symbol
symbol =
  Random.uniform Cherry [ Seven, Bar, Grapes ]

type alias Spin =
  { one : Symbol
  , two : Symbol
  , three : Symbol
  }

spin : Random.Generator Spin
spin =
  Random.map3 Spin symbol symbol symbol
```

We first create `Symbol` to describe the pictures that can appear on the slot machine. We then create a random generator that generates each symbol with equal probability.

From there we use `map3` to combine them into a new `spin` generator. It says to generate three symbols and then put them together into a `Spin`.

The point here is that from small building blocks, we can create a `Generator` that describes pretty complex behavior. And then from our application, we just have to say something like `Random.generate NewSpin spin` to get the next random value.

> **Exercises:** Here are a few ideas to make the example code on this page a bit more interesting!
>
> - Instead of showing a number, show the die face as an image.
> - Instead of showing an image of a die face, use `elm/svg` to draw it yourself.
> - Create a weighted die with `Random.weighted`.
> - Add a second die and have them both roll at the same time.
> - Have the dice flip around randomly before they settle on a final value.

Time

Clone the code or follow along in the online editor.

Now we are going to make a digital clock. (Analog will be an exercise!)

So far we have focused on commands. With the HTTP and randomness examples, we commanded Elm to do specific work immediately, but that is sort of a weird pattern for a clock. We *always* want to know the current time. This is where **subscriptions** come in!

After you read through the code, we will talk about how we are using the `elm/time` package here:

```
import Browser
import Html exposing (..)
import Task
import Time



-- MAIN


main =
  Browser.element
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }



-- MODEL


type alias Model =
  { zone : Time.Zone
  , time : Time.Posix
  }


init : () -> (Model, Cmd Msg)
init _ =
  ( Model Time.utc (Time.millisToPosix 0)
  , Task.perform AdjustTimeZone Time.here
  )



-- UPDATE


type Msg
  = Tick Time.Posix
  | AdjustTimeZone Time.Zone



update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Tick newTime ->
      ( { model | time = newTime }
      , Cmd.none
      )

    AdjustTimeZone newZone ->
      ( { model | zone = newZone }
      , Cmd.none
      )



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every 1000 Tick



-- VIEW


view : Model -> Html Msg
view model =
  let
    hour   = String.fromInt (Time.toHour   model.zone model.time)
    minute = String.fromInt (Time.toMinute model.zone model.time)
    second = String.fromInt (Time.toSecond model.zone model.time)
  in
  h1 [] [ text (hour ++ ":" ++ minute ++ ":" ++ second) ]
```

Let’s go through the new stuff.

`Time.Posix` and `Time.Zone`

To work with time successfully in programming, we need three different concepts:

- **Human Time** — This is what you see on clocks (8am) or on calendars (May 3rd). Great! But if my phone call is at 8am in Boston, what time is it for my friend in Vancouver? If it is at 8am in Tokyo, is that even the same day in New York? (No!) So between time zones based on ever-changing political boundaries and inconsistent use of daylight saving time, human time should basically never be stored in your `Model` or database! It is only for display!

- **POSIX Time** — With POSIX time, it does not matter where you live or what time of year it is. It is just the number of seconds elapsed since some arbitrary moment (in 1970). Everywhere you go on Earth, POSIX time is the same.

- **Time Zones** — A “time zone” is a bunch of data that allows you to turn POSIX time into human time. This is *not* just `UTC-7` or `UTC+3` though! Time zones are way more complicated than a simple offset! Every time Florida switches to DST forever or Samoa switches from UTC-11 to UTC+13, some poor soul adds a note to the IANA time zone database. That database is loaded onto every computer, and between POSIX time and all the corner cases in the database, we can figure out human times!

So to show a human being a time, you must always know `Time.Posix` and `Time.Zone`. That is it! So all that “human time” stuff is for the `view` function, not the `Model`. In fact, you can see that in our `view`:

```
view : Model -> Html Msg
view model =
  let
    hour   = String.fromInt (Time.toHour   model.zone model.time)
    minute = String.fromInt (Time.toMinute model.zone model.time)
    second = String.fromInt (Time.toSecond model.zone model.time)
  in
  h1 [] [ text (hour ++ ":" ++ minute ++ ":" ++ second) ]
```

The `Time.toHour` function takes `Time.Zone` and `Time.Posix` gives us back an `Int` from `0` to `23` indicating what hour it is in *your* time zone.

There is a lot more info about handling times in the README of `elm/time`. Definitely read it before doing more with time! Especially if you are working with scheduling, calendars, etc.

`subscriptions`

Okay, well how should we get our `Time.Posix` though? With a **subscription**!

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every 1000 Tick
```

We are using the `Time.every` function:

```
every : Float -> (Time.Posix -> msg) -> Sub msg
```

It takes two arguments:

1. A time interval in milliseconds. We said `1000` which means every second. But we could also say `60 * 1000` for every minute, or `5 * 60 * 1000` for every five minutes.
2. A function that turns the current time into a `Msg`. So every second, the current time is going to turn into a `Tick <time>` for our `update` function.

That is the basic pattern of any subscription. You give some configuration, and you describe how to produce `Msg` values. Not too bad!

`Task.perform`

Getting `Time.Zone` is a bit trickier. Our program created a **command** with:

```
Task.perform AdjustTimeZone Time.here
```

Reading through the `Task` docs is the best way to understand that line. The docs are written to actually explain the new concepts, and I think it would be too much of a digression to include a worse version of that info here. The point is just that we command the runtime to give us the `Time.Zone` wherever the code is running.

> **Exercises:**
>
> - Add a button to pause the clock, turning the `Time.every` subscription off.
> - Make the digital clock look nicer. Maybe add some `style` attributes.
> - Use `elm/svg` to make an analog clock with a red second hand!

JavaScript Interop

We have seen quite a bit of Elm so far! We learned **The Elm Architecture**. We learned about **types**. We learned how to interact with the outside world through **commands** and **subscriptions**. Things are going well!

But what happens when you need to do something in JavaScript? Maybe there is a JavaScript library you absolutely need? Maybe you want to embed Elm in an existing JavaScript application? Etc. This chapter will outline all the available options: flags, ports, and custom elements.

Whichever one you use, the first step is to initialize your Elm program.

Initializing Elm Programs

Running `elm make` produces HTML files by default. So if you say:

```
elm make src/Main.elm
```

It produces an `index.html` file that you can just open and start playing with. If you are getting into JavaScript interop, you want to produce JavaScript files instead:

```
elm make src/Main.elm --output=main.js
```

This produces a JavaScript file that exposes an `Elm.Main.init` function. So once you have `main.js` you can write your own HTML file that does whatever you want! For example:

```html
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="UTF-8">
  <title>Main</title>
  <link rel="stylesheet" href="whatever-you-want.css">
  <script src="main.js"></script>
</head>

<body>
  <div id="elm"></div>
  <script>
  var app = Elm.Main.init({
    node: document.getElementById('elm')
  });
  </script>
</body>
</html>
```

I want to call attention to a couple important lines here.

First, in the `<head>` of the document, you can load whatever you want! In our little example we loaded a CSS file called `whatever-you-want.css` :

```html
<link rel="stylesheet" href="whatever-you-want.css">
```

Maybe you write CSS by hand. Maybe you generate it somehow. Whatever the case, you can load it and use it in Elm. (There are some great options for specifying your CSS all *within* Elm as well, but that is a whole other topic!)

Second, we have a line to load our compiled Elm code:

```html
<script src="main.js"></script>
```

This will make an object called `Elm` available in global scope. So if you compile an Elm module called `Main` , you will have `Elm.Main` in JavaScript. If you compile an Elm module named `Home` , you will have `Elm.Home` in JavaScript. Etc.

Third, in the `<body>` of the document, we run a little bit of JavaScript code to initialize our Elm program:

```
<div id="elm"></div>
<script>
var app = Elm.Main.init({
  node: document.getElementById('elm')
});
</script>
```

We create an empty `<div>`. We want our Elm program to take over that node entirely. Maybe it is within a larger application, surrounded by tons of other stuff? That is fine!

The `<script>` tag then initializes our Elm program. We grab the `node` we want to take over, and give it to `Elm.Main.init` which starts our program.

Now that we can embed Elm programs in an HTML document, it is time to start exploring the three interop options: flags, ports, and web components!

Flags

The previous page showed the JavaScript needed to start an Elm program:

```
var app = Elm.Main.init({
  node: document.getElementById('elm')
});
```

It is possible to pass in some additional data though. For example, if we wanted to pass in the current time we could say:

```
var app = Elm.Main.init({
  node: document.getElementById('elm'),
  flags: Date.now()
});
```

We call this additional data `flags`. This allows you to customize the Elm program with all sorts of data!

> **Note:** This additional data is called "flags" because it is kind of like command line flags. You can call `elm make src/Main.elm`, but you can add some flags like `--optimize` and `--output=main.js` to customize its behavior. Same sort of thing.

Handling Flags

Just passing in JavaScript values is not enough. We need to handle them on the Elm side! The `Browser.element` function provides a way to handle flags with `init`:

```
element :
  { init : flags -> ( model, Cmd msg )
  , update : msg -> model -> ( model, Cmd msg )
  , subscriptions : model -> Subs msg
  , view : model -> Html msg
  }
  -> Program flags model msg
```

Notice that `init` has an argument called `flags`. So assuming we want to pass in the current time, we could write an `init` function like this:

```
init : Int -> ( Model, Cmd Msg )
init currentTime =
  ...
```

This means that Elm code gets immediate access to the flags you pass in from JavaScript. From there, you can put things in your model or run some commands. Whatever you need to do.

Verifying Flags

But what happens if `init` says it takes an `Int` flag, but someone tries to initialize with `Elm.Main.init({ flags: "haha, what now?" })` ?

Elm checks for that sort of thing, making sure the flags are exactly what you expect. Without this check, you could pass in anything, leading to runtime errors in Elm!

There are a bunch of types that can be given as flags:

- `Bool`
- `Int`
- `Float`
- `String`
- `Maybe`
- `List`
- `Array`
- tuples
- records
- `Json.Decode.Value`

Many folks always use a `Json.Decode.Value` because it gives them really precise control. They can write a decoder to handle any weird scenarios in Elm code, recovering from unexpected data in a nice way.

The other supported types actually come from before we had figured out a way to do JSON decoders. If you choose to use them, there are some subtleties to be aware of. The following examples show the desired flag type, and then the sub-points show what would happen with a couple different JS values:

- `init : Int -> ...`
    - `0` => `0`
    - `7` => `7`
    - `3.14` => error

    - `6.12` => error
- `init : Maybe Int -> ...`
    - `null` => `Nothing`
    - `42` => `Just 42`
    - `"hi"` => error
- `init : { x : Float, y : Float } -> ...`
    - `{ x: 3, y: 4, z: 50 }` => `{ x = 3, y = 4 }`
    - `{ x: 3, name: "tom" }` => error
    - `{ x: 360, y: "why?" }` => error
- `init : (String, Int) -> ...`
    - `['tom',42]` => `("Tom", 42)`
    - `["sue",33]` => `("Sue", 33)`
    - `["bob","4"]` => error
    - `['joe',9,9]` => error

Note that when one of the conversions goes wrong, **you get an error on the JS side!** We are taking the “fail fast” policy. Rather than the error making its way through Elm code, it is reported as soon as possible. This is another reason why people like to use `Json.Decode.Value` for flags. Instead of getting an error in JS, the weird value goes through a decoder, guaranteeing that you implement some sort of fallback behavior.

Ports

The previous two pages, we saw the JavaScript needed to start Elm programs and a way to pass in flags on initialization:

```
// initialize
var app = Elm.Main.init({
  node: document.getElementById('elm')
});

// initialize with flags
var app = Elm.Main.init({
  node: document.getElementById('elm'),
  flags: Date.now()
});
```

We can give information to the Elm program, but only when it starts. What if you want to talk to JavaScript while the program is running?

Message Passing

Elm allows you to pass messages between Elm and JavaScript through **ports**. Unlike the request/response pairs you see with HTTP, the messages sent through ports just go in one direction. It is like sending a letter. For example, banks in the United States send me hundreds of unsolicited letters, cajoling me to indebt myself to them so I will finally be happy. Those messages are all one-way. All letters are like that really. I may send a letter to my friend and she may reply, but there is nothing inherent about messages that demands request/response pairs. Point is, **Elm and JavaScript can communicate by sending these one-way messages through ports.**

Outgoing Messages

Say we want to use `localStorage` to cache some information. The solution is to set up a port that sends information out to JavaScript.

On the Elm side, this means defining the `port` like this:

```
port module Main exposing (..)

import Json.Encode as E

port cache : E.Value -> Cmd msg
```

The most important line is the `port` declaration. That creates a `cache` function, so we can create commands like `cache (E.int 42)` that will send a `Json.Encode.Value` out to JavaScript.

On the JavaScript side, we initialize the program like normal, but we then subscribe to all the outgoing `cache` messages:

```
var app = Elm.Main.init({
  node: document.getElementById('elm')
});
app.ports.cache.subscribe(function(data) {
  localStorage.setItem('cache', JSON.stringify(data));
});
```

Commands like `cache (E.int 42)` send values to anyone subscribing to the `cache` port in JavaScript. So the JS code would get `42` as `data` and cache it in `localStorage`.

In most programs that want to cache information like this, you communicate with JavaScript in two ways:

1. You pass in cached data through flags on initialization
2. You send data out periodically to update the cache

So there are only *outgoing* messages for this interaction with JS. And I would not get too intense trying to minimize the data crossing the border. Keep it simple, and be more tricky only if you find it necessary in practice!

> **Note 1:** This is not a binding to the `setItem` function! This is a common misinterpretation. **The point is not to cover the LocalStorage API one function at a time.** It is to ask for some caching. The JS code can decide to use LocalStorage, IndexedDB, WebSQL, or whatever else. So instead of thinking "should each JS function be a port?" think about "what needs to be accomplished in JS?" We have been thinking about caching, but it is the same in a fancy restaurant. You decide what you want, but you do not micromanage exactly how it is prepared. Your high-level message (your food order) goes back to the kitchen and you get a bunch of very specific messages back (drinks, appetizers, main course, desert, etc.) as a result. My point is that **well-designed ports create a clean separation of concerns.** Elm can do the view however it wants and JavaScript can do the caching however it wants.
>
> **Note 2:** There is not a LocalStorage package for Elm right now, so the current recommendation is to use ports like we just saw. Some people wonder about the timeline to get support directly in Elm. Some people wonder quite aggressively! I tried to write about that here.
>
> **Note 3:** Once you `subscribe` to outgoing port messages, you can `unsubscribe` as well. It works like `addEventListener` and `removeEventListener`, also requiring reference-equality of functions to work.

Incoming Messages

Say we are creating a chat room in JavaScript, and we are curious to try out Elm a bit. Pretty much every company that uses Elm today, started by converting just one element to try it out. Does it work nice? Does the team like it? If so, great, try more elements! If not, no big deal, revert and use the technologies that work best for you!

So when we look at our chat room app, we decide to convert an element that shows all active users. That means Elm needs to know about any changes to the active users list. Well, that sort of thing happens through ports!

On the Elm side, this means defining the `port` like this:

```
port module Main exposing (..)

import Json.Encode as E

type Msg
  = Searched String
  | Changed E.Value

port activeUsers : (E.Value -> msg) -> Sub msg
```

Again, the important line is the `port` declaration. It creates a `activeUsers` function, and if we subscribe to `activeUsers Changed`, we will get a `Msg` whenever folks send values in from JavaScript.

On the JavaScript side, we initialize the program like normal, but now we are able to send messages to any `activeUsers` subscriptions:

```
var activeUsers = // however this is defined

var app = Elm.Main.init({
  node: document.getElementById('elm'),
  flags: activeUsers
});

// after someone enters or exits
app.ports.activeUsers.send(activeUsers);
```

I start the Elm program with any known active users, and every time the active user list changes, I send the entire list through the `activeUsers` port.

Now you may be wondering, why send the *entire* list though? Why not just say who enters or exits? This approach sounds nice, but it creates the risk of synchronization errors. JavaScript thinks there are 20 active users, but somehow Elm thinks there are 25. Is there a bug in Elm code? Or in JavaScript? Forgot to send an exit message through ports? These bugs are extremely tricky to sort out, and you can end up wasting hours or days trying to figure them out.

Instead, I chose a design that makes synchronization errors impossible. JavaScript owns the state. All the Elm code does is get the complete list and display it. If the Elm code needs to change the list for some reason, it cannot! JavaScript owns the state. Instead, I would send a message out to JavaScript asking for specific changes. Point is, **state should be owned by Elm or by JavaScript, never both.** This dramatically reduces the risk of synchronization errors. Many folks who struggle with ports fall into this trap of never really deciding who owns the state. Be wary!

Notes

I want to add a couple notes about the examples we saw here:

- **All `port` declarations must appear in a `port module`.** It is probably best to organize all your ports into one `port module` so it is easier to see the interface all in one place.

- **Sending `Json.Decode.Value` through ports is recommended, but not the only way.** Like with flags, certain core types can pass through ports as well. This is from the time before JSON decoders, and you can read about it more here.
- **Ports are for applications.** A `port module` is available in applications, but not in packages. This ensures that application authors have the flexibility they need, but the package ecosystem is entirely written in Elm. I argued here that this will help us build a much stronger ecosystem and community in the long run.
- **Ports are about creating strong boundaries!** Definitely do not try to make a port for every JS function you need. You may really like Elm and want to do everything in Elm no matter the cost, but ports are not designed for that. Instead, focus on questions like "who owns the state?" and use one or two ports to send messages back and forth. If you are in a complex scenario, you can even simulate `Msg` values by sending JS like `{ tag: "active-users-changed", list: ... }` where you have a tag for all the variants of information you might send across.

I hope this information will help you find ways to embed Elm in your existing JavaScript! It is not as glamorous as doing a full-rewrite in Elm, but history has shown that it is a much more effective strategy.

> ## Aside: Design Considerations
>
> Ports are somewhat of an outlier in the history of languages. There are two common interop strategies, and Elm did neither of them:
>
> 1. **Full backwards compatibility.** For example, C++ is a superset of C, and TypeScript is a superset of JavaScript. This is the most permissive approach, and it has proven extremely effective. By definition, everyone is using your language already.
> 2. **Foreign function interface (FFI)** This allows direct bindings to functions in the host language. For example, Scala can call Java functions directly. Same with Clojure/Java, Python/C, Haskell/C, and many others. Again, this has proven quite effective.
>
> These paths are attractive, but they are not ideal for Elm for two main reasons:
>
> 1. **Losing Guarantees.** One of the best things about Elm is that there are entire categories of problems you just do not have to worry about, but if we can use JS directly in any package, all that goes away. Does this package produce runtime exceptions? When? Will it mutate the values I give to it? Do I need to detect that? Does the package have side-effects? Will it send messages to some 3rd party

> servers? A decent chunk of Elm users are drawn to the language specifically because they do not have to think like that anymore.
>
> 2. **Package Flooding.** There is quite high demand to directly copy JavaScript APIs into Elm. In the two years before `elm/html` existed, I am sure someone would have contributed jQuery bindings if it was possible. This has already happened in the typed functional languages that use more traditional interop designs. As far as I know, package flooding is unique to compile-to-JS languages. The pressure is not nearly as high in Python for example, so I think that downside is a product of the unique culture and history of the JavaScript ecosystem.
>
> Given these pitfalls, ports are attractive because they let you get things done in JavaScript while preserving the best parts of Elm. Great! On the flipside, it means Elm cannot piggyback on the JS ecosystem to gain more libraries more quickly. If you take a longer-view, I think this is actually a key strength. As a result:
>
> 1. **Packages are designed for Elm.** As members of the Elm community get more experience and confidence, we are starting to see fresh approaches to layout and data visualization that work seamlessly with The Elm Architecture and the overall ecosystem. I expect this to keep happening with other sorts of problems!
> 2. **Packages are portable.** If the compiler someday produces x86 or WebAssembly, the whole ecosystem just keeps working, but faster! Ports guarantee that all packages are written entirely in Elm, and Elm itself was designed such that other non-JS compiler targets are viable.
>
> So this is definitely a longer and harder path, but languages live for 30+ years. They have to support teams and companies for decades, and when I think about what Elm will look like in 20 or 30 years, I think the tradeoffs that come with ports look really promising! My talk What is Success? starts a little slow, but it gets into this a bit more!

Custom Elements

On the last few pages, we have seen (1) how to start Elm programs from JavaScript, (2) how to pass data in as flags on initialization, and (3) how to send messages between Elm and JS with ports. But guess what people? There is another way to do interop!

Browsers seem to be supporting custom elements more and more, and that turns out to be quite helpful for embedding JS into Elm programs.

I am not particularly experienced with this technique yet, so I will defer to Luke. He has a lot more experience, and I think his Elm Europe talk is an excellent introduction!

Video link

I would love to provide text and examples here as well, and I hope to come back to this page in a calmer time and write it up myself. But in the meantime, I hope the community can fill the gap with blog posts!

Web Apps

So far we have been creating Elm programs with `Browser.element`, allowing us to take over a single node in a larger application. This is great for *introducing* Elm at work (as described here) but what happens after that? How can we use Elm more extensively?

In this chapter, we will learn how to create a “web app” with a bunch of different pages that all integrate nicely with each other, but we must start by controlling a single page.

Control the Document

The first step is to switch to starting programs with `Browser.document`:

```
document :
  { init : flags -> ( model, Cmd msg )
  , view : model -> Document msg
  , update : msg -> model -> ( model, Cmd msg )
  , subscriptions : model -> Sub msg
  }
  -> Program flags model msg
```

The arguments are almost exactly the same as `Browser.element`, except for the `view` function. Rather than returning an `Html` value, you return a `Document` like this:

```
type alias Document msg =
  { title : String
  , body : List (Html msg)
  }
```

This gives you control over the `<title>` and the `<body>` of the document. Perhaps your program downloads some data and that helps you determine a more specific title. Now you can just change it in your `view` function!

Serve the Page

The compiler produces HTML by default, so you can compile your code like this:

```
elm make src/Main.elm
```

The output will be a file named `index.html` that you can serve like any other HTML file. That works fine, but you can get a bit more flexibility by (1) compiling Elm to JavaScript and (2) making your own custom HTML file. To take that path, you compile like this:

```
elm make src/Main.elm --output=main.js
```

This will produce `main.js` which you can load from a custom HTML file like this:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="UTF-8">
  <title>Main</title>
  <link rel="stylesheet" href="whatever-you-want.css">
  <script src="main.js"></script>
</head>
<body>
  <script>var app = Elm.Main.init();</script>
</body>
</html>
```

This HTML is pretty simple. You load whatever you need in the `<head>` and you initialize your Elm program in the `<body>`. The Elm program will take it from there and render everything.

Either way, now you have some HTML that you can send to browsers. You can get that HTML to people with free services like GitHub Pages or Netlify, or maybe you make your own server and run a VPS with a service like Digital Ocean. Whatever works for you! You just need a way to get HTML into a browser.

> **Note 1:** Creating custom HTML is helpful if you are doing something custom with CSS. Many people use projects like `rtfeldman/elm-css` to handle all of their styles from within Elm, but maybe you are working in a team where there is lots of predefined CSS. Maybe the team is even using one of those CSS preprocessors. That is all fine. Just load the final CSS file in the `<head>` of your HTML file.
>
> **Note 2:** The Digital Ocean link above is a referral link, so if you sign up through that and end up using the service, we get a $25 credit towards our hosting costs for `elm-lang.org` and `package.elm-lang.org`.

Navigation

We just saw how to serve one page, but say we are making a website like `package.elm-lang.org`. It has a bunch of pages (e.g. search, README, docs) that all work differently. How does it do that?

Multiple Pages

The simple way would be to serve a bunch of different HTML files. Going to the home page? Load new HTML. Going to `elm/core` docs? Load new HTML. Going to `elm/json` docs? Load new HTML.

Until Elm 0.19, that is exactly what the package website did! It works. It is simple. But it has some weaknesses:

1. **Blank Screens.** The screen goes white everytime you load new HTML. Can we do a nice transition instead?
2. **Redundant Requests.** Each package has a single `docs.json` file, but it gets loaded each time you visit a module like `String` or `Maybe`. Can we share the data between pages somehow?
3. **Redundant Code.** The home page and the docs share a lot of functions, like `Html.text` and `Html.div`. Can this code be shared between pages?

We can improve all three cases! The basic idea is to only load HTML once, and then be a bit tricky to handle URL changes.

Single Page

Instead of creating our program with `Browser.element` or `Browser.document`, we can create a `Browser.application` to avoid loading new HTML when the URL changes:

```
application :
  { init : flags -> Url -> Key -> ( model, Cmd msg )
  , view : model -> Document msg
  , update : msg -> model -> ( model, Cmd msg )
  , subscriptions : model -> Sub msg
  , onUrlRequest : UrlRequest -> msg
  , onUrlChange : Url -> msg
  }
  -> Program flags model msg
```

It extends the functionality of `Browser.document` in three important scenarios.

When the application starts, `init` gets the current `Url` from the browsers navigation bar. This allows you to show different things depending on the `Url`.

When someone clicks a link, like `<a href="/home">Home</a>`, it is intercepted as a `UrlRequest`. So instead of loading new HTML with all the downsides, `onUrlRequest` creates a message for your `update` where you can decide exactly what to do next. You can save scroll position, persist data, change the URL yourself, etc.

When the URL changes, the new `Url` is sent to `onUrlChange`. The resulting message goes to `update` where you can decide how to show the new page.

So rather than loading new HTML, these three additions give you full control over URL changes. Let’s see it in action!

Example

We will start with the baseline `Browser.application` program. It just keeps track of the current URL. Skim through the code now! Pretty much all of the new and interesting stuff happens in the `update` function, and we will get into those details after the code:

```
import Browser
import Browser.Navigation as Nav
import Html exposing (..)
import Html.Attributes exposing (..)
import Url



-- MAIN


main : Program () Model Msg
main =
  Browser.application
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    , onUrlChange = UrlChanged
    , onUrlRequest = LinkClicked
    }



-- MODEL
```

```
type alias Model =
  { key : Nav.Key
  , url : Url.Url
  }


init : () -> Url.Url -> Nav.Key -> ( Model, Cmd Msg )
init flags url key =
  ( Model key url, Cmd.none )



-- UPDATE


type Msg
  = LinkClicked Browser.UrlRequest
  | UrlChanged Url.Url


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    LinkClicked urlRequest ->
      case urlRequest of
        Browser.Internal url ->
          ( model, Nav.pushUrl model.key (Url.toString url) )

        Browser.External href ->
          ( model, Nav.load href )

    UrlChanged url ->
      ( { model | url = url }
      , Cmd.none
      )



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions _ =
  Sub.none



-- VIEW


view : Model -> Browser.Document Msg
view model =
  { title = "URL Interceptor"
  , body =
      [ text "The current URL is: "
      , b [] [ text (Url.toString model.url) ]
      , ul []
          [ viewLink "/home"
          , viewLink "/profile"
          , viewLink "/reviews/the-century-of-the-self"
          , viewLink "/reviews/public-opinion"
          , viewLink "/reviews/shah-of-shahs"
          ]
      ]
  }


viewLink : String -> Html msg
viewLink path =
  li [] [ a [ href path ] [ text path ] ]
```

The `update` function can handle either `LinkClicked` or `UrlChanged` messages. There is a lot of new stuff in the `LinkClicked` branch, so we will focus on that first!

`UrlRequest`

Whenever someone clicks a link like `<a href="/home">/home</a>`, it produces a `UrlRequest` value:

```
type UrlRequest
  = Internal Url.Url
  | External String
```

The `Internal` variant is for any link that stays on the same domain. So if you are browsing `https://example.com`, internal links include things like `settings#privacy`, `/home`, `https://example.com/home`, and `//example.com/home`.

The `External` variant is for any link that goes to a different domain. Links like `https://elm-lang.org/examples`, `https://static.example.com`, and `http://example.com/home` all go to a different domain. Notice that changing the protocol from `https` to `http` is considered a different domain!

Whichever link someone presses, our example program is going to create a `LinkClicked` message and send it to the `update` function. That is where we see most of the interesting new code!

`LinkClicked`

Most of our `update` logic is deciding what to do with these `UrlRequest` values:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    LinkClicked urlRequest ->
      case urlRequest of
        Browser.Internal url ->
          ( model, Nav.pushUrl model.key (Url.toString url) )

        Browser.External href ->
          ( model, Nav.load href )

    UrlChanged url ->
      ( { model | url = url }
      , Cmd.none
      )
```

The particularly interesting functions are `Nav.load` and `Nav.pushUrl`. These are both from the `Browser.Navigation` module which is all about changing the URL in different ways. We are using the two most common functions from that module:

- `load` loads all new HTML. It is equivalent to typing the URL into the URL bar and pressing enter. So whatever is happening in your `Model` will be thrown out, and a whole new page is loaded.
- `pushUrl` changes the URL, but does not load new HTML. Instead it triggers a `UrlChanged` message that we handle ourselves! It also adds an entry to the “browser history” so things work normal when people press the `BACK` or `FORWARD` buttons.

So looking back at the `update` function, we can understand how it all fits together a bit better now. When the user clicks a `https://elm-lang.org` link, we get an `External` message and use `load` to load new HTML from those servers. But when the user clicks a `/home` link, we get an `Internal` message and use `pushUrl` to change the URL *without* loading new HTML!

> **Note 1:** Both `Internal` and `External` links are producing commands immediately in our example, but that is not required! When someone clicks an `External` link, maybe you want to save textbox content to your database before navigating away. Or when someone clicks an `Internal` link, maybe you want to use `getViewport` to save the scroll position in case they navigate `BACK` later. That is all possible! It is a normal `update` function, and you can delay the navigation and do whatever you want.
>
> **Note 2:** If you want to restore "what they were looking at" when they come `BACK`, scroll position is not perfect. If they resize their browser or reorient their device, it could be off by quite a lot! So it is probably better to save "what they were looking at" instead. Maybe that means using `getViewportOf` to figure out exactly what is on screen at the moment. The particulars depend on how your application works exactly, so I cannot give exact advice!

`UrlChanged`

There are a couple ways to get `UrlChanged` messages. We just saw that `pushUrl` produces them, but pressing the browser `BACK` and `FORWARD` buttons produce them as well. And like I was saying in the notes a second ago, when you get a `LinkClicked` message, the `pushUrl` command may not be given immediately.

So the nice thing about having a separate `UrlChanged` message is that it does not matter how or when the URL changed. All you need to know is that it did!

We are just storing the new URL in our example here, but in a real web app, you need to parse the URL to figure out what content to show. That is what the next page is all about!

> **Note:** I skipped talking about `Nav.Key` to try to focus on more important concepts. But I will explain here for those who are interested!
>
> A navigation `Key` is needed to create navigation commands (like `pushUrl`) that change the URL. You only get access to a `Key` when you create your program with `Browser.application`, guaranteeing that your program is equipped to detect these URL changes. If `Key` values were available in other kinds of programs, unsuspecting programmers would be sure to run into some annoying bugs and learn a bunch of techniques the hard way!
>
> As a result of all that, we have a line in our `Model` for our `Key`. A relatively low price to pay to help everyone avoid an extremely subtle category of problems!

Parsing URLs

In a realistic web app, we want to show different content for different URLs:

- `/search`
- `/search?q=seiza`
- `/settings`

How do we do that? We use the `elm/url` to parse the raw strings into nice Elm data structures. This package makes the most sense when you just look at examples, so that is what we will do!

Example 1

Say we have an art website where the following addresses should be valid:

- `/topic/architecture`
- `/topic/painting`
- `/topic/sculpture`
- `/blog/42`
- `/blog/123`
- `/blog/451`
- `/user/tom`
- `/user/sue`
- `/user/sue/comment/11`
- `/user/sue/comment/51`

So we have topic pages, blog posts, user information, and a way to look up individual user comments. We would use the `Url.Parser` module to write a URL parser like this:

```
import Url.Parser exposing (Parser, (</>), int, map, oneOf, s, string)

type Route
  = Topic String
  | Blog Int
  | User String
  | Comment String Int

routeParser : Parser (Route -> a) a
routeParser =
  oneOf
    [ map Topic   (s "topic" </> string)
    , map Blog    (s "blog" </> int)
    , map User    (s "user" </> string)
    , map Comment (s "user" </> string </> s "comment" </> int)
    ]

-- /topic/pottery        ==>  Just (Topic "pottery")
-- /topic/collage        ==>  Just (Topic "collage")
-- /topic/               ==>  Nothing

-- /blog/42              ==>  Just (Blog 42)
-- /blog/123             ==>  Just (Blog 123)
-- /blog/mosaic          ==>  Nothing

-- /user/tom/            ==>  Just (User "tom")
-- /user/sue/            ==>  Just (User "sue")
-- /user/bob/comment/42  ==>  Just (Comment "bob" 42)
-- /user/sam/comment/35  ==>  Just (Comment "sam" 35)
-- /user/sam/comment/    ==>  Nothing
-- /user/                ==>  Nothing
```

The `Url.Parser` module makes it quite concise to fully turn valid URLs into nice Elm data!

Example 2

Now say we have a personal blog where addresses like this are valid:

- `/blog/12/the-history-of-chairs`
- `/blog/13/the-endless-september`
- `/blog/14/whale-facts`
- `/blog/`
- `/blog?q=whales`
- `/blog?q=seiza`

In this case we have individual blog posts and a blog overview with an optional query parameter. We need to add the `Url.Parser.Query` module to write our URL parser this time:

```
import Url.Parser exposing (Parser, (</>), (<?>), int, map, oneOf, s, string)
import Url.Parser.Query as Query

type Route
  = BlogPost Int String
  | BlogQuery (Maybe String)

routeParser : Parser (Route -> a) a
routeParser =
  oneOf
    [ map BlogPost  (s "blog" </> int </> string)
    , map BlogQuery (s "blog" <?> Query.string "q")
    ]

-- /blog/14/whale-facts  ==>  Just (BlogPost 14 "whale-facts")
-- /blog/14              ==>  Nothing
-- /blog/whale-facts     ==>  Nothing
-- /blog/                ==>  Just (BlogQuery Nothing)
-- /blog                 ==>  Just (BlogQuery Nothing)
-- /blog?q=chabudai      ==>  Just (BlogQuery (Just "chabudai"))
-- /blog/?q=whales       ==>  Just (BlogQuery (Just "whales"))
-- /blog/?query=whales   ==>  Just (BlogQuery Nothing)
```

The `</>` and `<?>` operators let us to write parsers that look quite like the actual URLs we want to parse. And adding `Url.Parser.Query` allowed us to handle query parameters like `?q=seiza`.

Example 3

Okay, now we have a documentation website with addresses like this:

- `/Basics`
- `/Maybe`
- `/List`
- `/List#map`
- `/List#filter`
- `/List#foldl`

We can use the `fragment` parser from `Url.Parser` to handle these addresses like this:

```
type alias Docs =
  (String, Maybe String)

docsParser : Parser (Docs -> a) a
docsParser =
  map Tuple.pair (string </> fragment identity)

-- /Basics     ==>  Just ("Basics", Nothing)
-- /Maybe      ==>  Just ("Maybe", Nothing)
-- /List       ==>  Just ("List", Nothing)
-- /List#map   ==>  Just ("List", Just "map")
-- /List#      ==>  Just ("List", Just "")
-- /List/map   ==>  Nothing
-- /           ==>  Nothing
```

So now we can handle URL fragments as well!

Synthesis

Now that we have seen a few parsers, we should look at how this fits into a `Browser.application` program. Rather than just saving the current URL like last time, can we parse it into useful data and show that instead?

```
TODO
```

The major new things are:

1. Our `update` parses the URL when it gets a `UrlChanged` message.
2. Our `view` function shows different content for different addresses!

It is really not too fancy. Nice!

But what happens when you have 10 or 20 or 100 different pages? Does it all go in this one `view` function? Surely it cannot be all in one file. How many files should it be in? What should be the directory structure? That is what we will discuss next!

Modules

Elm has **modules** to help you grow your codebase in a nice way. On the most basic level, modules let you break your code into multiple files.

Defining Modules

Elm modules work best when you define them around a central type. Like how the `List` module is all about the `List` type. So say we want to build a module around a `Post` type for a blogging website. We can create something like this:

```
module Post exposing (Post, estimatedReadTime, encode, decoder)

import Json.Decode as D
import Json.Encode as E


-- POST

type alias Post =
  { title : String
  , author : String
  , content : String
  }


-- READ TIME

estimatedReadTime : Post -> Float
estimatedReadTime post =
  toFloat (wordCount post) / 220

wordCount : Post -> Int
wordCount post =
  List.length (String.words post.content)


-- JSON

encode : Post -> E.Value
encode post =
  E.object
    [ ("title", E.string post.title)
    , ("author", E.string post.author)
    , ("content", E.string post.content)
    ]

decoder : D.Decoder Post
decoder =
  D.map3 Post
    (D.field "title" D.string)
    (D.field "author" D.string)
    (D.field "content" D.string)
```

The only new syntax here is that `module Post exposing (..)` line at the very top. That means the module is known as `Post` and only certain values are available to outsiders. As written, the `wordCount` function is only available *within* the `Post` module. Hiding functions like this is one of the most important techniques in Elm!

> **Note:** If you forget to add a module declaration, Elm will use this one instead:
>
> ```
> module Main exposing (..)
> ```
>
> This makes things easier for beginners working in just one file. They should not be confronted with the module system on their first day!

Growing Modules

As your application gets more complex, you will end up adding things to your modules. It is normal for Elm modules to be in the 400 to 1000 line range, as I explain in The Life of a File. But when you have multiple modules, how do you decide *where* to add new code?

I try to use the following heuristics when code is:

- **Unique** — If logic only appears in one place, I break out top-level helper functions as close to the usage as possible. Maybe use a comment header like `-- POST PREVIEW` to indicate that the following definitions are related to previewing posts.
- **Similar** — Say we want to show `Post` previews on the home page and on the author pages. On the home page, we want to emphasize the interesting content, so we want longer snippets. But on the author page, we want to emphasize the breadth of content, so we want to focus on titles. These cases are *similar*, not the same, so we go back to the **unique** heuristic. Just write the logic separately.
- **The Same** — At some point we will have a bunch of **unique** code. That is fine! But perhaps we find that some definitions contain logic that is *exactly* the same. Break out a helper function for that logic! If all the uses are in one module, no need to do anything more. Maybe put a comment header like `-- READ TIME` if you really want.

These heuristics are all about making helper functions within a single file. You only want to create a new module when a bunch of these helper functions all center around a specific custom type. For example, you start by creating a `Page.Author` module, and do not create a `Post` module until the helper functions start piling up. At that point, creating a new module should make your code feel easier to navigate and understand. If it does not, go back to the version that was clearer. More modules is not more better! Take the path that keeps the code simple and clear.

To summarize, assume **similar** code is **unique** by default. (It usually is in user interfaces in the end!) If you see logic that is **the same** in different definitions, make some helper functions with appropriate comment headers. When you have a bunch of helper functions about a specific type, *consider* making a new module. If a new module makes your code clearer, great! If not, go back. More files is not inherently simpler or clearer.

> **Note:** One of the most common ways to get tripped up with modules is when something that was once **the same** becomes **similar** later on. Very common, especially in user interfaces! Folks will often try to create a Frankenstein function that handles all the different cases. Adding more arguments. Adding more *complex* arguments. The better path is to accept that you now have two **unique** situations and copy the code into both places. Customize it exactly how you need. Then see if any of the resulting logic is **the same**. If so, move it out into helpers. **Your long functions should split into multiple smaller functions, not grow longer and more complex!**

Using Modules

It is customary in Elm for all of your code to live in the `src/` directory. That is the default for `elm.json` even. So our `Post` module would need to live in a file named `src/Post.elm`. From there, we can `import` a module and use its exposed values. There are four ways to do that:

```elm
import Post
-- Post.Post, Post.estimatedReadTime, Post.encode, Post.decoder

import Post as P
-- P.Post, P.estimatedReadTime, P.encode, P.decoder

import Post exposing (Post, estimatedReadTime)
-- Post, estimatedReadTime
-- Post.Post, Post.estimatedReadTime, Post.encode, Post.decoder

import Post as P exposing (Post, estimatedReadTime)
-- Post, estimatedReadTime
-- P.Post, P.estimatedReadTime, P.encode, P.decoder
```

I recommend using `exposing` pretty rarely. Ideally on zero or one of your imports. Otherwise, it can start getting hard to figure out where things came from when reading though. “Wait, where is `filterPostBy` from again? What arguments does it take?” It gets harder and harder to read through code as you add more `exposing`. I tend to use it for `import Html exposing (..)` but not on anything else. For everything else, I recommend using the standard `import` and maybe using `as` if you have a particularly long module name!

Structuring Web Apps

Like I was saying on the previous page, **all modules should be built around a central type.** So if I was making a web app for blog posts, I would start with modules like this:

- `Main`
- `Page.Home`
- `Page.Search`
- `Page.Author`

I would have a module for each page, centered around the `Model` type. Those modules follow The Elm Architecture with the typical `Model`, `init`, `update`, `view`, and whatever helper functions you need. From there, I would just keep growing those modules longer and longer. Keep adding the types and functions you need. If I ever notice that I created a custom type with a couple helper functions, I *might* move that out into its own module.

Before we see some examples, I want to emphasize an important strategy.

Do Not Plan Ahead

Notice that my `Page` modules do not make any guesses about the future. I do not try to define modules that can be used in multiple places. I do not try to share any functions. This is on purpose!

Early in my projects, I always have these grand schemes of how everything will fit together. “The pages for editing and viewing posts both care about posts, so I will have a `Post` module!” But as I write my application, I find that only the viewing page should have a publication date. And I actually need to track editing differently to cache data when tabs are closed. And they actually need to be stored a bit differently on servers as a result. Etc. I end up turning `Post` into a big mess to handle all these competing concerns, and it ends up being worse for both pages.

By just starting with pages, it becomes much easier to see when things are **similar**, but not **the same**. The norm in user interfaces! So with editing and viewing posts, it seems plausible that we could end up with an `EditablePost` type and a `ViewablePost` type, each with different structure, helper functions, and JSON decoders. Maybe those types are complex enough to warrant their own module. Maybe not! I would just write the code and see what happens.

This works because the compiler makes it really easy to do huge refactors. If I realize I got something majorly wrong across 20 files, I just fix it.

Examples

You can see examples of this structure in the following open-source projects:

- `elm-spa-example`
- `package.elm-lang.org`

> ## Aside: Culture Shock
>
> Folks coming from JavaScript tend to bring habits, expectations, and anxieties that are specific to JavaScript. They are legitimately important in that context, but they can cause some pretty severe troubles when transferred to Elm.
>
> ### Defensive Instincts
>
> In The Life of a File I point out some JavaScript Folk Knowledge that leads you astray in Elm:
>
> - ~~**“Prefer shorter files.”**~~ In JavaScript, the longer your file is, the more likely you have some sneaky mutation that will cause a really difficult bug. But in Elm, that is not possible! Your file can be 2000 lines long and that still cannot happen.
> - ~~**“Get architecture right from the beginning.”**~~ In JavaScript, refactoring is extremely risky. In many cases, it is cheaper just to rewrite it from scratch. But in Elm, refactoring is cheap and reliable! You can make changes in 20 different files with confidence.
>
> These defensive instincts are protecting you from problems that do not exist in Elm. Knowing this in your mind is different than knowing it in your gut though, and I have observed that JS folks often feel deeply uncomfortable when they see files pass the 400 or 600 or 800 line mark. **So I encourage you to push your limit on number of lines!** See how far you can go. Try using comment headers, try making helper functions, but keep it all in one file. Having this experience yourself is extremely valuable!
>
> ### MVC

> Some folks see The Elm Architecture and have the intuition to divide their code into separate modules for `Model`, `Update`, and `View`. Do not do this!
>
> It leads to unclear and debatable boundaries. What happens when `Post.estimatedReadTime` is used in both the `update` and `view` functions? Totally reasonable, but it does not clearly *belong* to one or the other. Maybe you need a `Utils` module? Maybe it actually is a controller kind of thing? The resulting code tends to be hard to navigate because placing each function is now an ontological question, and all of your colleagues have different theories. What is an `estimatedReadTime` really? What is its essence? Estimation? What would Richard think is its essence? Time?
>
> **If you build each module around a type, you rarely run into these kinds of questions.** You have a `Page.Home` module that contains your `Model`, `update`, and `view`. You write helper functions. You add a `Post` type eventually. You add an `estimatedReadTime` function. Maybe someday there are a bunch of helpers about that `Post` type, and maybe it is worth splitting into its own module. With this convention, you end up spending a lot less time considering and reconsidering module boundaries. I find that the code also comes out much clearer.
>
> ## Components
>
> Folks coming from React expect everything to be components. **Actively trying to make components is a recipe for disaster in Elm.** The root issue is that components are objects:
>
> - components = local state + methods
> - local state + methods = objects
>
> It would be odd to start using Elm and wonder "how do I structure my application with objects?" There are no objects in Elm! Folks in the community would recommend using custom types and functions instead.
>
> Thinking in terms of components encourages you create modules based on the visual design of your application. “There is a sidebar, so I need a `Sidebar` module.” It would be way easier to just make a `viewSidebar` function and pass it whatever arguments it needs. It probably does not even have any state. Maybe one or two fields? Just put it in the `Model` you already have. If it really is worth splitting out into its own module, you will know because you will have a custom type with a bunch of relevant helper functions!
>
> Point is, writing a `viewSidebar` function **does not** mean you need to create a corresponding `update` and `Model` to go with it. Resist this instinct. **Just write the helper functions you need.**

Optimization

There are two major types of optimization in Elm. Optimizing performance and optimizing asset size:

- **Performance** — The slowest thing in browsers is the DOM. By a huge margin. I have done a lot of profiling to speed up Elm applications, and most things have no noticable impact. Using better data structures? Negligible. Caching the results of computations in my model? Negligible *and* my code is worse now. The only thing that makes a big difference is using `Html.Lazy` and `Html.Keyed` to do fewer DOM operations.
- **Asset Size** — Running in browsers means we have to care about download times. The smaller we can get our assets, the faster they load on mobile devices and slow internet connections. This is probably more important than any of the performance optimizations you will do! Fortunately, the Elm compiler does a really good job of making your code as small as possible, so you do not need to do a bunch of work making your code confusing to get decent outcomes here.

Both are important though, so this chapter will go through how this all works!

`Html.Lazy`

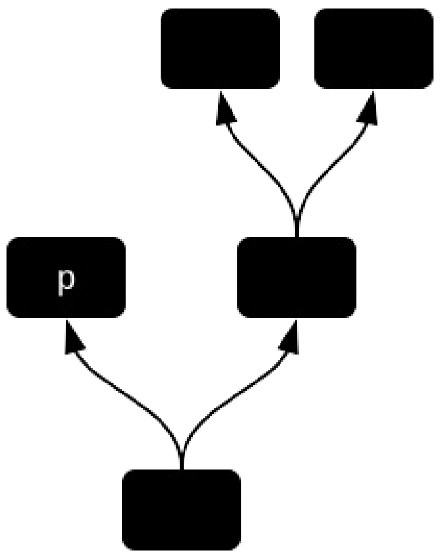
In the `elm/html` package is used to show things on screen. To understand how to optimize it, we need to learn how it works in the first place!

What is the DOM?

If you are creating an HTML file, you would write HTML directly like this:

```html
<div>
  <p>Chair alternatives include:</p>
  <ul>
    <li>seiza</li>
    <li>chabudai</li>
  </ul>
</div>
```

You can think of this as producing some DOM data structure behind the scenes:



The black boxes represent heavy-weight DOM objects with hundreds of attributes. And when any of them change, it can trigger expensive renders and reflows of page content.

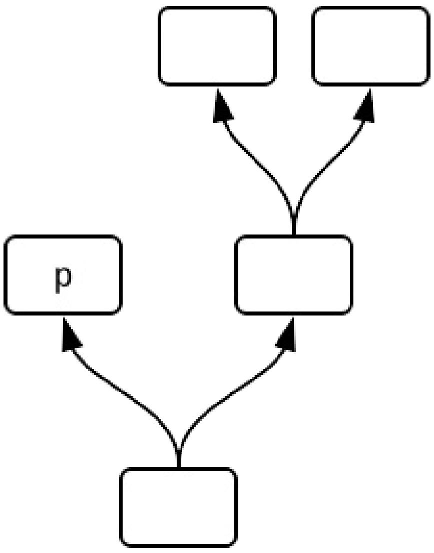
What is Virtual DOM?

If you are creating an Elm file, you would use `elm/html` to write something like this:

```
viewChairAlts : List String -> Html msg
viewChairAlts chairAlts =
  div []
    [ p [] [ text "Chair alternatives include:" ]
    , ul [] (List.map viewAlt chairAlts)
    ]

viewAlt : String -> Html msg
viewAlt chairAlt =
  li [] [ text chairAlt ]
```

You can think of `viewChairAlts ["seiza","chabudai"]` as producing some “Virtual DOM” data structure behind the scenes:



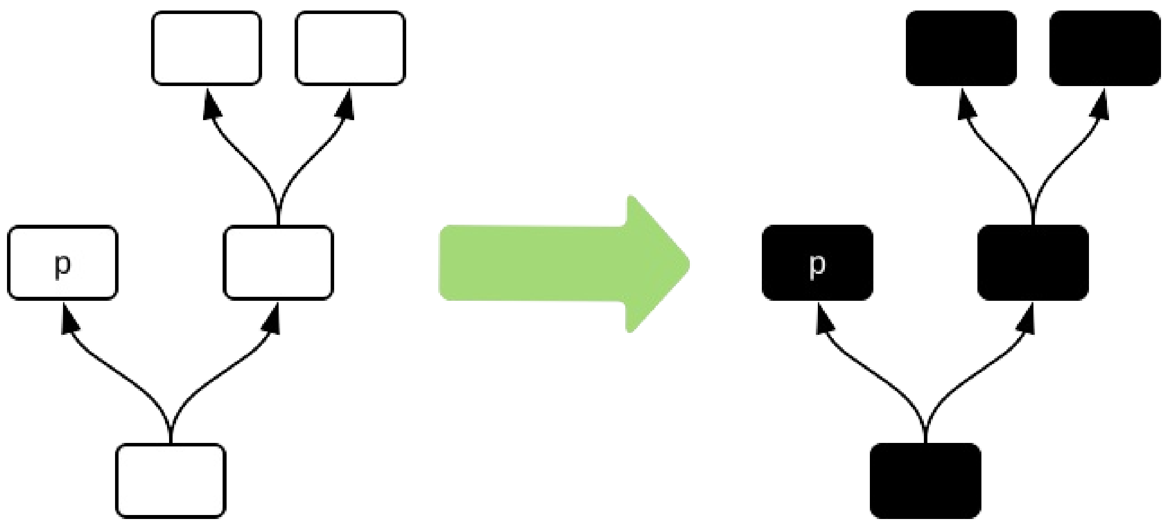
The white boxes represent light-weight JavaScript objects. They only have the attributes you specify. Their creation can never cause renders or reflows. Point is, compared to DOM nodes, these are much cheaper to allocate!

Render

If we are always working with these virtual nodes in Elm, how does it get converted to the DOM we see on screen? When an Elm program starts, it goes like this:

- Call `init` to get the initial `Model` .
- Call `view` to get the initial virtual nodes.

Now that we have virtual nodes, we make an exact replica in the real DOM:

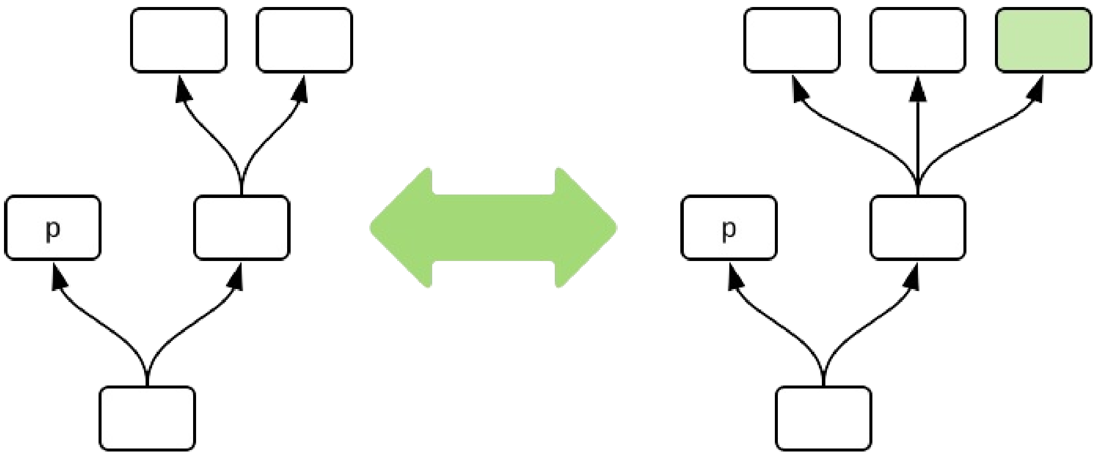


Great! But what about when things change? Redoing the whole DOM on every frame does not work, so what do we do instead?

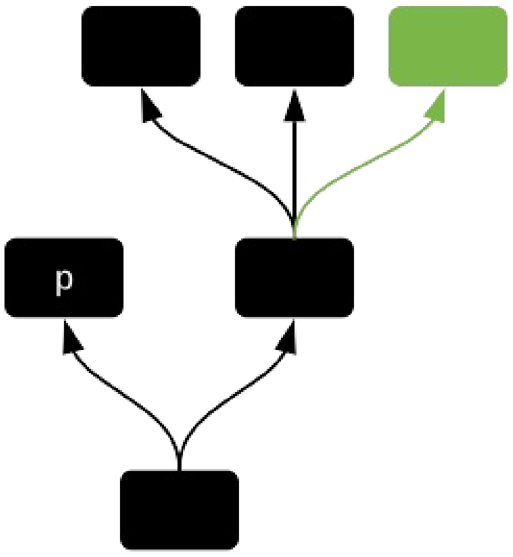
Diffing

Once we have the initial DOM, we switch to working primarily with virtual nodes instead. Whenever the `Model` changes, we run `view` again. From there, we "diff" the resulting virtual nodes to figure out how to touch the DOM as little as possible.

So imagine our `Model` gets a new chair alternative, and we want to add a new `li` node for it. Behind the scenes, Elm diffs the **current** virtual nodes and the **next** virtual nodes to detect any changes:



It noticed that a third `li` was added. I marked it in green. Elm now knows exactly how to modify the real DOM to make it match. Just insert that new `li`:



This diffing process makes it possible to touch the DOM as little as possible. And if no differences are found, we do not need to touch the DOM at all! So this process helps minimize the renders and reflows that need to happen.

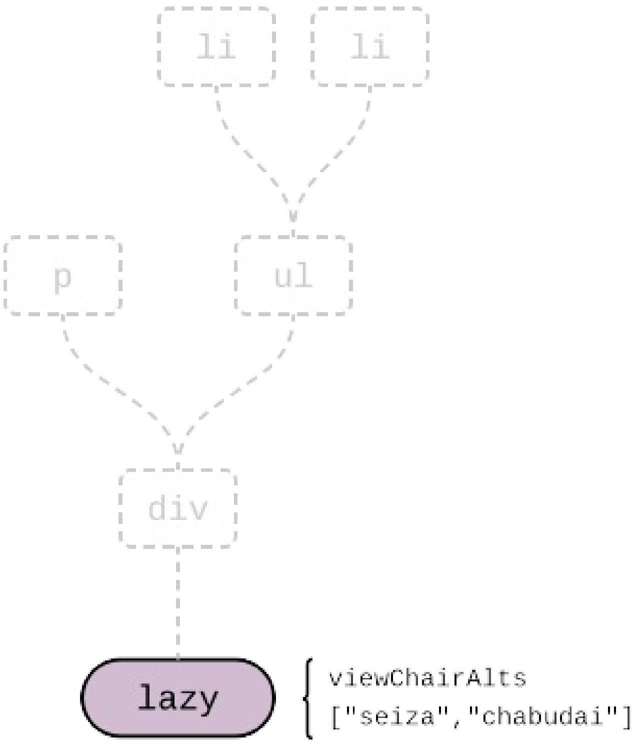
But can we do even less work?

`Html.Lazy`

The `Html.Lazy` module makes it possible to not even build the virtual nodes! The core idea is the `lazy` function:

```
lazy : (a -> Html msg) -> a -> Html msg
```

Going back to our chair example, we called `viewChairAlts ["seiza","chabudai"]`, but we could just as easily have called `lazy viewChairAlts ["seiza","chabudai"]` instead. The lazy version allocates a single “lazy” node like this:



The node just keeps a reference to the function and arguments. Elm can put the function and arguments together to generate the whole structure if needed, but it is not always needed!

One of the cool things about Elm is the “same input, same output” guarantee for functions. So whenever we run into two “lazy” nodes while diffing virtual nodes, we ask is the function the same? Are the arguments the same? If they are all the same, we know the resulting virtual nodes are the same as well! **So we can skip building the virtual nodes entirely!** If any of them have changed, we can build the virtual nodes and do a normal diff.

> **Note:** When are two values “the same” though? To optimize for performance, we use JavaScript’s `===` operator behind the scenes:
>
> - Structural equality is used for `Int`, `Float`, `String`, `Char`, and `Bool`.
> - Reference equality is used for records, lists, custom types, dictionaries, etc.
>
> Structural equality means that `4` is the same as `4` no matter how you produced those values. Reference equality means the actual pointer in memory has to be the same. Using reference equality is always cheap `O(1)`, even when the data structure has thousands or millions of entries. So this is mostly about making sure that using `lazy` will never slow your code down a bunch by accident. All the checks are super cheap!

Usage

The ideal place to put a lazy node is at the root of your application. Many applications are set up to have distinct visual regions like headers, sidebars, search results, etc. And when people are messing with one, they are very rarely messing with the others. This creates really natural lines for `lazy` calls!

For example, in my TodoMVC implementation, the `view` is defined like this:

```
view : Model -> Html Msg
view model =
  div
    [ class "todomvc-wrapper"
    , style "visibility" "hidden"
    ]
    [ section
        [ class "todoapp" ]
        [ lazy viewInput model.field
        , lazy2 viewEntries model.visibility model.entries
        , lazy2 viewControls model.visibility model.entries
        ]
    , infoFooter
    ]
```

Notice that the text input, entries, and controls are all in separate lazy nodes. So I can type however many characters I want in the input without ever building virtual nodes for the entries or controls. They are not changing! So the first tip is **try to use lazy nodes at the root of your application.**

It can also be useful to use lazy in long lists of items. In the TodoMVC app, it is all about adding entries to your todo list. You could conceivable have hundreds of entries, but they change very infrequently. This is a great candidate for laziness! By switching `viewEntry entry` to `lazy viewEntry entry` we can skip a bunch of allocation that is very rarely useful. So the second tip is **try to use lazy nodes on repeated structures where each individual item changes infrequently.**

Summary

Touching the DOM is way more expensive than anything that happens in a normal user interface. Based on my benchmarking, you can do whatever you want with fancy data structures, but in the end it only matters how much you successfully use `lazy`.

On the next page, we will learn a technique to use `lazy` even more!

`Html.Keyed`

On the previous page, we learned how Virtual DOM works and how we can use `Html.Lazy` to avoid a bunch of work. Now we are going to introduce `Html.Keyed` to skip even more work.

This optimization is particularly helpful for lists of data in your interface that must support **insertion**, **removal**, and **reordering**.

The Problem

Say we have a list of all the Presidents of the United States. And maybe it lets us sort by name, by education, by net worth, and by birthplace.

When the diffing algorithm (described on the previous page) gets to a long list of items, it just goes through pairwise:

- Diff the current 1st element with the next 1st element.
- Diff the current 2nd element with the next 2nd element.
- ...

But when you change the sort order, all of these are going to be different! So you end up doing a lot of work on the DOM when you could have just shuffled some nodes around.

This issue exists with insertion and removal as well. Say you remove the 1st of 100 items. Everything is going to be off-by-one and look different. So you get 99 diffs and one removal at the end. No good!

The Solution

The fix for all of this is `Html.Keyed.node`, which makes it possible to pair each entry with a “key” that easily distinguishes it from all the others.

So in our presidents example, we could write our code like this:

```
import Html exposing (..)
import Html.Keyed as Keyed
import Html.Lazy exposing (lazy)

viewPresidents : List President -> Html msg
viewPresidents presidents =
  Keyed.node "ul" [] (List.map viewKeyedPresident presidents)

viewKeyedPresident : President -> (String, Html msg)
viewKeyedPresident president =
  ( president.name, lazy viewPresident president )

viewPresident : President -> Html msg
viewPresident president =
  li [] [ ... ]
```

Each child node is associated with a key. So instead of doing a pairwise diff, we can diff based on matching keys!

Now the Virtual DOM implementation can recognize when the list is resorted. It first matches all the presidents up by key. Then it diffs those. We used `lazy` for each entry, so we can skip all that work. Nice! It then figures out how to shuffle the DOM nodes to show things in the order you want. So the keyed version does a lot less work in the end.

Resorting helps show how it works, but it is not the most common case that really needs this optimization. **Keyed nodes are extremely important for insertion and removal.** When you remove the 1st of 100 elements, using keyed nodes allows the Virtual DOM implementation to recognize that immediately. So you get a single removal instead of 99 diffs.

Summary

Touching the DOM is extraordinarily slow compared to the sort of computations that happen in a normal application. **Always reach for `Html.Lazy` and `Html.Keyed` first.** I recommend verifying this with profiling as much as possible. Some browsers provide a timeline view of your program, like this. It gives you a summary of how much time is spent in loading, scripting, rendering, painting, etc. If you see that 10% of the time is spent scripting, you could make your Elm code twice as fast and not make any noticable difference. Whereas simple additions of lazy and keyed nodes could start taking big chunks out of that other 90% by touching the DOM less!

Asset Size

The only thing that is slower than touching the DOM is talking to servers. Especially for folks on mobile phones with slow internet. So you can optimize your code all day with `Html.Lazy` and `Html.Keyed`, but your application will still feel slow if it loads slowly!

A great way to improve is to send fewer bits. For example, if a 122kb asset can become a 9kb asset, it will load faster! We get results like that by using the following techniques:

- **Compilation.** The Elm compiler can perform optimizations like dead code elimination and record field renaming. So it can cut unused code and shorten record field names like `userStatus` in the generated code.
- **Minification.** In the JavaScript world, there are tools called “minifiers” that do a bunch of transformations. They shorten variables. They inline. They convert `if` statements to ternary operators. They turn `'\u0041'` to `'A'`. Anything to save a few bits!
- **Compression.** Once you have gotten the code as small as possible, you can use a compression algorithm like gzip to shrink it even further. It does particularly well with keywords like `function` and `return` that you just cannot get rid of in the code itself.

Elm makes it pretty easy to get all this set up for your project. No need for some complex build system. It is just two terminal commands!

Instructions

Step one is to compile with the `--optimize` flag. This does things like shortening record field names.

Step two is to minify the resulting JavaScript code. I use a minifier called `uglifyjs`, but you can use a different one if you want. The neat thing about `uglifyjs` is all its special flags. These flags unlock optimizations that are unreliable in normal JS code, but thanks to the design of Elm, they are totally safe for us!

Putting those together, we can optimize `src/Main.elm` with two terminal commands:

```
elm make src/Main.elm --optimize --output=elm.js
uglifyjs elm.js --compress 'pure_funcs="F2,F3,F4,F5,F6,F7,F8,F9,A2,A3,A4,A5,A6,A7,A8,A
9",pure_getters,keep_fargs=false,unsafe_comps,unsafe' | uglifyjs --mangle --output=elm
.min.js
```

After this you will have an `elm.js` and a smaller `elm.min.js` file!

> **Note 1:** `uglifyjs` is called twice there. First to `--compress` and second to `--mangle`. This is necessary! Otherwise `uglifyjs` will ignore our `pure_funcs` flag.
>
> **Note 2:** If the `uglifyjs` command is not available in your terminal, you can run the command `npm install uglify-js --global` to download it. If you do not have `npm` either, you can get it with nodejs.

Scripts

It is hard to remember all those flags for `uglifyjs`, so it is probably better to write a script that does this.

Say we want a bash script that produces `elm.js` and `elm.min.js` files. On Mac or Linux, we can define `optimize.sh` like this:

```
#!/bin/sh

set -e

js="elm.js"
min="elm.min.js"

elm make --optimize --output=$js $@

uglifyjs $js --compress 'pure_funcs="F2,F3,F4,F5,F6,F7,F8,F9,A2,A3,A4,A5,A6,A7,A8,A9",
pure_getters,keep_fargs=false,unsafe_comps,unsafe' | uglifyjs --mangle --output=$min

echo "Compiled size:$(cat $js | wc -c) bytes  ($js)"
echo "Minified size:$(cat $min | wc -c) bytes  ($min)"
echo "Gzipped size: $(cat $min | gzip -c | wc -c) bytes"
```

Now if I run `./optimize.sh src/Main.elm` on my TodoMVC code, I see something like this in the terminal:

```
Compiled size:  122297 bytes  (elm.js)
Minified size:   24123 bytes  (elm.min.js)
Gzipped size:     9148 bytes
```

Pretty neat! We only need to send about 9kb to get this program to people!

The important commands here are `elm` and `uglifyjs` which work on any platform, so it should not be too tough to do something similar on Windows.

Advice

I recommend writing a `Browser.application` and compiling to a single JavaScript file as we have seen here. It will get downloaded (and cached) when people first visit. Elm creates quite small files compared to the popular competitors, as you can see here, so this strategy can take you quite far.

> **Note:** In theory, it is possible to get even smaller assets with Elm. It is not possible right now, but if you are working on 50k lines of Elm or more, we would like to learn about your situation as part of a user study. More details here!

Next Steps

We have a bunch of foundational knowledge now. The next steps are all about gaining experience and building relationships.

Build Something

Experience is a great teacher, so I recommend building an **application** that interests you. If you do not have something in mind, maybe something like this:

- **Expand Examples** — Take some of the examples from this book and add to them. The code is here! Mess with `elm-spa-example`.
- **Something from Work** — Maybe there is something you do at work, and you want to see how it might work in Elm. Try it out on your own and see how it goes! This will set you up well for the advice in How to Use Elm at Work.
- **Data Visualization** — Use a package like `terezka/line-charts` to display data that interests you. I would start with some fake data, but maybe work up to trying to showing data from somewhere else. There is a ton of economic and health data available that would benefit from better presentation!
- **Games** — I got into programming by making games like pong, breakout, and space invaders. Maybe you will like that sort of thing too! Start by drawing stuff with `elm/svg`. From there, try responding to events like `onKeyDown`, `onMouseMove`, and `onAnimationFrame`. At some point, you can even get into 3D graphics with `elm-explorations/webgl`!

Ask Questions

There are loads of friendly and knowledgable folks on Slack and Discourse. Whether you just started programming or have 20 years experience in industry, the #beginners channel on Slack is great for people new to programming in Elm! Maybe you have an error message you are stuck on? Maybe you are struggling to understand JSON decoders? Maybe the `Task` type is tripping you up? Maybe you are curious to get some feedback on a custom type you defined? **Whatever the problem, you can always ask for help!**

Meet People

There are meetups all over the world. We encourage organizers to run code nights where folks can build projects and relationships. The obvious benefit is that you can get help with whatever you are working on, but you also meet everyone else who is using Elm in town. Maybe someone is working on something cool that inspires you. Maybe you learn a technique you did not know you did not know. Maybe someone has a job opening. Maybe knowing a bunch of local Elm programmers will help your case at work. Maybe it is just a fun time. Programmers tend to undervalue the benefits of these personal relationships, but it is one of the most important parts of a healthy programming language community!

Types as Sets

We have seen primitive types like `Bool` and `String`. We have made our own custom types like this:

```
type Color = Red | Yellow | Green
```

One of the most important techniques in Elm programming is to make **the possible values in code** exactly match **the valid values in real life**. This leaves no room for invalid data, and this is why I always encourage folks to focus on custom types and data structures.

In pursuit of this goal, I have found it helpful to understand the relationship between types and sets. It sounds like a stretch, but it really helps develop your mindset!

Sets

You can think of types as a set of values.

- `Bool` is the set `{ True, False }`
- `Color` is the set `{ Red, Yellow, Green }`
- `Int` is the set `{ ... -2, -1, 0, 1, 2 ... }`
- `Float` is the set `{ ... 0.9, 0.99, 0.999 ... 1.0 ... }`
- `String` is the set `{ "", "a", "aa", "aaa" ... "hello" ... }`

So when you say `x : Bool` it is like saying `x` is in the `{ True, False }` set.

Cardinality

Some interesting things happen when you start figuring out how many values are in these sets. For example the `Bool` set `{ True, False }` contains two values. So math people would say that `Bool` has a cardinality of two. So conceptually:

- cardinality(`Bool`) = 2
- cardinality(`Color`) = 3
- cardinality(`Int`) = ∞
- cardinality(`Float`) = ∞
- cardinality(`String`) = ∞

This gets more interesting when we start thinking about types like `(Bool, Bool)` that combine sets together.

> **Note:** The cardinality for `Int` and `Float` are actually smaller than infinity. Computers need to fit the numbers into a fixed amount of bits (as described here) so it is more like cardinality(`Int32`) = 2^32 and cardinality(`Float32`) = 2^32. The point is just that it is a lot.

Multiplication (Tuples and Records)

When you combine types with tuples, the cardinalities get multiplied:

- cardinality(`(Bool, Bool)`) = cardinality(`Bool`) × cardinality(`Bool`) = 2 × 2 = 4
- cardinality(`(Bool, Color)`) = cardinality(`Bool`) × cardinality(`Color`) = 2 × 3 = 6

To make sure you believe this, try listing all the possible values of `(Bool, Bool)` and `(Bool, Color)` . Do they match the numbers we predicted? How about for `(Color, Color)` ?

But what happens when we use infinite sets like `Int` and `String` ?

- cardinality(`(Bool, String)`) = 2 × ∞
- cardinality(`(Int, Int)`) = ∞ × ∞

I personally really like the idea of having two infinities. One wasn't enough? And then seeing infinite infinities. Aren't we going to run out at some point?

> **Note:** So far we have used tuples, but records work exactly the same way:
>
> - cardinality(`(Bool, Bool)`) = cardinality(`{ x : Bool, y : Bool }`)
> - cardinality(`(Bool, Color)`) = cardinality(`{ active : Bool, color : Color }`)
>
> And if you define `type Point = Point Float Float` then cardinality(`Point`) is equivalent to cardinality(`(Float, Float)`). It is all multiplication!

Addition (Custom Types)

When figuring out the cardinality of a custom type, you add together the cardinality of each variant. Let's start by looking at some `Maybe` and `Result` types:

- cardinality(`Result Bool Color`) = cardinality(`Bool`) + cardinality(`Color`) = 2 + 3 = 5
- cardinality(`Maybe Bool`) = 1 + cardinality(`Bool`) = 1 + 2 = 3
- cardinality(`Maybe Int`) = 1 + cardinality(`Int`) = 1 + ∞

To persuade yourself that this is true, try listing out all the possible values in the `Maybe Bool` and `Result Bool Color` sets. Does it match the numbers we got?

Here are some other examples:

```
type Height
  = Inches Int
  | Meters Float

-- cardinality(Height)
-- = cardinality(Int) + cardinality(Float)
-- = ∞ + ∞


type Location
  = Nowhere
  | Somewhere Float Float

-- cardinality(Location)
-- = 1 + cardinality((Float, Float))
-- = 1 + cardinality(Float) × cardinality(Float)
-- = 1 + ∞ × ∞
```

Looking at custom types this way helps us see when two types are equivalent. For example, `Location` is equivalent to `Maybe (Float, Float)`. Once you know that, which one should you use? I prefer `Location` for two reasons:

1. The code becomes more self-documenting. No need to wonder if `Just (1.6, 1.8)` is a location or a pair of heights.
2. The `Maybe` module may expose functions that do not make sense for my particular data. For example, combining two locations probably should not work like `Maybe.map2`. Should one `Nowhere` mean that everything is `Nowhere`? Seems weird!

In other words, I write a couple lines of code that are *similar* to other code, but it gives me a level of clarity and control that is extremely valuable for large code bases and teams.

Who Cares?

Thinking of “types as sets” helps explain an important class of bugs: **invalid data**. For example, say we want to represent the color of a traffic light. The set of valid values are { red, yellow, green } but how do we represent that in code? Here are three different approaches:

- `type alias Color = String` — We could decide that `"red"`, `"yellow"`, `"green"` are the three strings we will use, and that all the other ones are *invalid data*. But what happens if invalid data is produced? Maybe someone makes a typo like `"rad"`. Maybe someone types `"RED"` instead. Should all functions have checks for incoming color arguments? Should all functions have tests to make sure color results are valid? The root issue is that cardinality(`Color`) = ∞, meaning there are (∞ - 3) invalid values. We will need to do a lot of checking to make sure none of them ever show up!

- `type alias Color = { red : Bool, yellow : Bool, green : Bool }` — The idea here is that the idea of “red” is represented by `Color True False False`. But what about `Color True True True`? What does it mean for it to be all the colors at once? This is *invalid data*. Just like with the `String` representation, we end up writing checks in our code and tests to make sure there are no mistakes. In this case, cardinality(`Color`) = 2 × 2 × 2 = 8, so there are only 5 invalid values. There are definitely fewer ways to mess up, but we should still have some checks and tests.

- `type Color = Red | Yellow | Green` — In this case, invalid data is impossible. cardinality(`Color`) = 1 + 1 + 1 = 3, exactly corresponding to the set of three values in real life. So there is no point checking for invalid color data in our code or tests. It cannot exist!

So the whole point here is that **ruling out invalid data makes your code shorter, simpler, and more reliable.** By making sure the set of *possible* values in code exactly matches the set of *valid* values in real life, many problems just go away. This is a sharp knife!

As your program changes, the set of possible values in code may start to diverge from the set of valid values in real life. **I highly recommend revisiting your types periodically to make them match again.** This is like noticing your knife has become dull and sharpening it with a whetstone. This kind of maintenance is a core part of programming in Elm.

When you start thinking this way, you end up needing fewer tests, yet having more reliable code. You start using fewer dependencies, yet accomplishing things more quickly. Similarly, someone skilled with a knife probably will not buy a SlapChop. There is definitely a place for blenders and food processors, but it is smaller than you might think. No one runs ads about how you can be independent and self-sufficient without any serious downsides. No money in that!

Aside on Language Design

Thinking of types as sets like this can also be helpful in explaining why a language would feel “easy” or “restrictive” or “error-prone” to some people. For example:

- **Java** — There are primitive values like `Bool` and `String`. From there, you can create classes with a fixed set of fields of different types. This is much like records in Elm, allowing you to multiply cardinalities. But it is quite difficult to do addition. You can do it with subtyping, but it is quite an elaborate process. So where `Result Bool Color` is easy in Elm, it is pretty tough in Java. I think some people find Java “restrictive” because designing a type with cardinality 5 is quite difficult, often seeming like it is not worth the trouble.

- **JavaScript** — Again, there are primitive values like `Bool` and `String`. From there you can create objects with a dynamic set of fields, allowing you to multiply cardinalities. This is much more lightweight than creating classes. But like Java, doing addition is not particularly easy. For example, you can simulate `Maybe Int` with objects like `{ tag: "just", value: 42 }` and `{ tag: "nothing" }`, but this is really still multiplication of cardinality. This makes it quite difficult to exactly match the set of valid values in real life. So I think people find JavaScript “easy” because designing a type with cardinality ($\infty \times \infty \times \infty$) is super easy and that can cover pretty much anything, but other people find it “error-prone” because designing a type with cardinality 5 is not really possible, leaving lots of space for invalid data.

Interestingly, some imperative languages have custom types! Rust is a great example. They call them enums to build on the intuition folks may have from C and Java. So in Rust, addition of cardinalities is just as easy as in Elm, and it brings all the same benefits!

I think the point here is that “addition” of types is extraordinarily underrated in general, and thinking of “types as sets” helps clarify why certain language designs would produce certain frustrations.

Types as Bits

There are all sorts of types in Elm:

- `Bool`
- `Int`
- `String`
- `(Int, Int)`
- `Maybe Int`
- ...

We have a conceptual understanding of them by now, but how are they understood by a computer? How is `Maybe Int` stored on a hard disk?

Bits

A **bit** is little box that has two states. Zero or a one. On or off. Computer memory is one super long sequence of bits.

Okay, so all we have is a bunch of bits. Now we need to represent *everything* with that!

`Bool`

A `Bool` value can be either `True` or `False`. This corresponds exactly to a bit!

`Int`

An `Int` value is some whole number like `0`, `1`, `2`, etc. You cannot fit that in a single bit, so the only other option is to use multiple bits. So normally, an `Int` would be a sequence of bits, like these:

```
00000000
00000001
00000010
00000011
...
```

We can arbitrarily assign meaning to each of these sequences. So maybe `00000000` is zero and `00000001` is one. Great! We can just start assigning numbers to bit sequences in ascending order. But eventually we will run out of bits...

By some quick math, eight bits only allow (2^8 = 256) numbers. What about perfectly reasonable numbers like 9000 and 8004?

The answer is to just add more bits. For a long time, people used 32 bits. That allowed for (2^32 = 4,294,967,296) numbers which covers the kinds of numbers humans typically think about. Computers these days support 64-bit integers, allowing for (2^64 = 18,446,744,073,709,551,616) numbers. That is a lot!

> **Note:** If you are curious how addition works, learn about two’s complement. It reveals that numbers are not assigned to bit sequences arbitrarily. For the sake of making addition as fast as possible, this particular way of assigning numbers works really well.

`String`

The string `"abc"` is the sequence of characters `a` `b` `c`, so we will start by trying to represent characters as bits.

One of the early ways of encoding characters is called ASCII. Just like with integers, they decided to list out a bunch of bit sequences and start assigning values arbitrarily:

```
00000000
00000001
00000010
00000011
...
```

So every character needed to fit in eight bits, meaning only 256 characters could be represented! But if you only care about English, this actually works out pretty well. You need to cover 26 lower-case letters, 26 upper-case letters, and 10 numbers. That is 62. There is a bunch of room left for symbols and other weird stuff. You can see what they ended up with here.

We have an idea for characters now, but how will the computer know where the `String` ends and the next piece of data begins? It is all just bits. Characters look just like `Int` values really! So we need some way to mark the end.

These days, languages tend to do this by storing the **length** of the string. So a string like `"hello"` might look something like `5` `h` `e` `l` `l` `o` in memory. So you know a `String` always starts with 32-bits representing the length. And whether the length is 0 or

9000, you know exactly where the characters end.

> **Note:** At some point, folks wanted to cover languages besides English. This effort eventually resulted in the UTF-8 encoding. It is quite brilliant really, and I encourage you to learn about it. It turns out that "get the 5th character" is harder than it sounds!

`(Int, Int)`

What about tuples? Well, `(Int, Int)` is two `Int` values, and each one is a sequence of bits. Let's just put those two sequences next to each other in memory and call it a day!

Custom Types

A custom type is all about combining different types. Those different types may have all sorts of different shapes. We will start with the `Color` type:

```
type Color = Red | Yellow | Green
```

We can assign each case a number: `Red = 0` , `Yellow = 1` , and `Green = 2` . Now we can use the `Int` representation. Here we only need two bits to cover all the possible cases, so `00` is red, `01` is yellow, `10` is green, and `11` is unused.

But what about custom types that hold additional data? Like `Maybe Int` ? The typical approach is to set aside some bits to "tag" the data, so we can decide that `Nothing = 0` and `Just = 1` . Here are some examples:

- `Nothing` = `0`
- `Just 12` = `1` `00001100`
- `Just 16` = `1` `00010000`

A `case` expression always looks at that "tag" before deciding what to do next. If it sees a `0` it knows there is no more data. If it sees a `1` it knows it is followed by a sequence of bits representing the data.

This "tag" idea is similar to putting the length at the beginning of `String` values. The values may be different sizes, but the code can always figure out where they start and end.

Summary

Eventually, all values need to be represented in bits. This page gives a rough overview of how that actually works.

Normally there is no real reason to think about this, but I found it to be helpful in deepening my understanding of custom types and `case` expressions. I hope it is helpful to you as well!

> **Note:** If you think this is interesting, it may be fun to learn more about garbage collection. I have found The Garbage Collection Handbook to be an excellent resource on the topic!

Function Types

As you look through packages like `elm/core` and `elm/html`, you will definitely see functions with multiple arrows. For example:

```
String.repeat : Int -> String -> String
String.join : String -> List String -> String
```

Why so many arrows? What is going on here?!

Hidden Parentheses

It starts to become clearer when you see all the parentheses. For example, it is also valid to write the type of `String.repeat` like this:

```
String.repeat : Int -> (String -> String)
```

It is a function that takes an `Int` and then produces *another* function. So if we go into `elm repl` we can see this in action:

```
> String.repeat 4
<function> : String -> String

> String.repeat 4 "ha"
"hahahaha" : String

> String.join "|"
<function> : List String -> String

> String.join "|" ["red","yellow","green"]
"red|yellow|green" : String
```

So conceptually, **every function accepts one argument.** It may return another function that accepts one argument. Etc. At some point it will stop returning functions.

We *could* always put the parentheses to indicate that this is what is really happening, but it starts to get pretty unwieldy when you have multiple arguments. It is the same logic behind writing `4 * 2 + 5 * 3` instead of `(4 * 2) + (5 * 3)`. It means there is a bit extra to learn, but it is so common that it is worth it.

Fine, but what is the point of this feature in the first place? Why not do `(Int, String) -> String` and give all the arguments at once?

Partial Application

It is quite common to use the `List.map` function in Elm programs:

```
List.map : (a -> b) -> List a -> List b
```

It takes two arguments: a function and a list. From there it transforms every element in the list with that function. Here are some examples:

- `List.map String.reverse ["part","are"] == ["trap","era"]`
- `List.map String.length ["part","are"] == [4,3]`

Now remember how `String.repeat 4` had type `String -> String` on its own? Well, that means we can say:

- `List.map (String.repeat 2) ["ha","choo"] == ["haha","choochoo"]`

The expression `(String.repeat 2)` is a `String -> String` function, so we can use it directly. No need to say `(\str -> String.repeat 2 str)`.

Elm also uses the convention that **the data structure is always the last argument** across the ecosystem. This means that functions are usually designed with this possible usage in mind, making this a pretty common technique.

Now it is important to remember that **this can be overused!** It is convenient and clear sometimes, but I find it is best used in moderation. So I always recommend breaking out top-level helper functions when things get even a *little* complicated. That way it has a clear name, the arguments are named, and it is easy to test this new helper function. In our example, that means creating:

```
-- List.map reduplicate ["ha","choo"]

reduplicate : String -> String
reduplicate string =
  String.repeat 2 string
```

This case is really simple, but (1) it is now clearer that I am interested in the linguistic phenomenon known as reduplication and (2) it will be quite easy to add new logic to `reduplicate` as my program evolves. Maybe I want shm-reduplication support at some point?

In other words, **if your partial application is getting long, make it a helper function.** And if it is multi-line, it should *definitely* be turned into a top-level helper! This advice applies to using anonymous functions too.

> **Note:** If you are ending up with "too many" functions when you use this advice, I recommend using comments like `-- REDUPLICATION` to give an overview of the next five or ten functions. Old school! I have shown this with `-- UPDATE` and `-- VIEW` comments in previous examples, but it is a generic technique that I use in all my code. And if you are worried about files getting too long with this advice, I recommend watching The Life of a File!

Pipelines

Elm also has a pipe operator that relies on partial application. For example, say we have a `sanitize` function for turning user input into integers:

```
-- BEFORE
sanitize : String -> Maybe Int
sanitize input =
  String.toInt (String.trim input)
```

We can rewrite it like this:

```
-- AFTER
sanitize : String -> Maybe Int
sanitize input =
  input
    |> String.trim
    |> String.toInt
```

So in this "pipeline" we pass the input to `String.trim` and then that gets passed along to `String.toInt`.

This is neat because it allows a "left-to-right" reading that many people like, but **pipelines can be overused!** When you have three or four steps, the code often gets clearer if you break out a top-level helper function. Now the transformation has a name. The arguments are named. It has a type annotation. It is much more self-documenting that way, and your teammates and your future self will appreciate it! Testing the logic gets easier too.

> **Note:** I personally prefer the `BEFORE`, but perhaps that is just because I learned functional programming in languages without pipes!