FREE eBook

LEARNING
Elm Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elm

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elm-language

It is an unofficial and free Elm Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Elm Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Elm Language

Remarks

[Elm][1] is a friendly functional programming language compiling to JavaScript. Elm focuses on browser-based GUIs, single-page applications.

Users usually praise it for:

- No runtime exceptions.
- Best compiler errors ever
- The ease of refactoring.
- Expressive type system
- The Elm Architecture, which Redux is inspired by.

Versions

Version	Release Date
0.18.0	2016-11-14
0.17.1	2016-06-27
0.17	2016-05-10
0.16	2015-11-19
0.15.1	2015-06-30
0.15	2015-04-20

Examples

Installation

To start development with Elm, you need to install a set of tools called elm-platform.

It includes: elm-make, elm-reactor, elm-repl and elm-package.

All of these tools are available through CLI, in other words you can use them from your terminal.

Pick one of the following methods to install Elm:

Using the installer

Download the installer from the official website and follow the installation wizard.

Using npm

You can use Node Package Manager to install Elm platform.

Global installation:

```
$ npm install elm -g
```

Local installation:

```
$ npm install elm
```

Locally installed Elm platform tools are accessible via:

```
$ ./node_modules/.bin/elm-repl  # launch elm-repl from local node_modules/
```

Using homebrew

```
$ brew install elm
```

Switch between versions with elm-use

Install elm-use

```
$ npm install -g elm-use
```

Switch to an older or newer elm version

```
$ elm-use 0.18  // or whatever version you want to use
```

Further reading

Learn how to Initialize and build your first project.

Hello World

See how to compile this code in Initialize and build

```
import Html
```

```
main = Html.text "Hello World!"
```

Editors

Atom

- https://atom.io/packages/language-elm
- https://atom.io/packages/elmjutsu

Light Table

- https://github.com/rundis/elm-light

Sublime Text

- https://packagecontrol.io/packages/Elm%20Language%20Support

Vim

- https://github.com/ElmCast/elm-vim

Emacs

- https://github.com/jcollard/elm-mode

IntelliJ IDEA

- https://plugins.jetbrains.com/plugin/8192

Brackets

- https://github.com/tommot348/elm-brackets

VS Code

- https://marketplace.visualstudio.com/items?itemName=sbrink.elm

Initialize and build

You should have Elm platform installed on your computer, the following tutorial is written with the assumption, that you are familiar with terminal.

Initialization

Create a folder and navigate to it with your terminal:

```
$ mkdir elm-app
$ cd elm-app/
```

Initialize Elm project and install core dependencies:

```
$ elm-package install -y
```

`elm-package.json` and `elm-stuff` folder should appear in your project.

Create the entry point for your application `Main.elm` and paste Hello World example in to it.

Building the project

To build your first project, run:

```
$ elm-make Main.elm
```

This will produce `index.html` with the `Main.elm` file (and all dependencies) compiled into JavaScript and inlined into the HTML. **Try and open it in your browser!**

If this fails with the error `I cannot find module 'Html'.` it means that you are not using the latest version of Elm. You can solve the problem either by upgrading Elm and redoing the first step, or with the following command:

```
$ elm-package install elm-lang/html -y
```

In case you have your own `index.html` file (eg. when working with ports), you can also compile your Elm files to a JavaScript file:

```
$ elm-make Main.elm --output=elm.js
```

More info in the example Embedding into HTML.

Style Guide and elm-format

The official style guide is located on the homepage and generally goes for:

- readability (instead of compactness)
- ease of modification
- clean diffs

This means that, for example, this:

```
homeDirectory : String
homeDirectory =
  "/root/files"


evaluate : Boolean -> Bool
evaluate boolean =
  case boolean of
    Literal bool ->
        bool

    Not b ->
        not (evaluate b)

    And b b' ->
        evaluate b && evaluate b'

    Or b b' ->
        evaluate b || evaluate b'
```

is considered **better** than:

```
homeDirectory = "/root/files"

eval boolean = case boolean of
    Literal bool -> bool
    Not b        -> not (eval b)
    And b b'     -> eval b && eval b'
    Or b b'      -> eval b || eval b'
```

0.16

The tool elm-format helps by formatting your source code for you **automatically** (typically on save), in a similar vein to Go language's gofmt. Again, the underlying value is having **one consistent style** and saving arguments and flamewars about various issues like *tabs vs. spaces* or *indentation length*.

You can install `elm-format` following the instructions on the Github repo. Then configure your editor to format the Elm files automatically or run `elm-format FILE_OR_DIR --yes` manually.

Embedding into HTML

There are three possibilities to insert Elm code into a existing HTML page.

Embed into the body tag

Supposing you have compiled the Hello World example into `elm.js` file, you can let Elm take over the `<body>` tag like so:

```
<!DOCTYPE html>
<html>
    <body>
        <script src="elm.js"></script>
        <script>
          Elm.Main.fullscreen()
        </script>
    </body>
</html>
```

WARNING: Sometimes some chrome extensions mess with `<body>` which can cause your app to break in production. It's recommended to always embed in a specific div. More info here.

Embed into a Div (or other DOM node)

Alternatively, by providing concrete HTML element, Elm code can be run in that specific page element:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <div id='app'></div>
        <script src="elm.js"></script>
        <script>
            Elm.Main.embed(document.getElementById('app'))
        </script>
    </body>
</html>
```

Embed as a Web worker (no UI)

Elm code can also be started as a worker and communicate thru ports:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello Worker</title>
    </head>
    <body>
        <script src="elm.js"></script>
        <script>
            var app = Elm.Main.worker();
            app.ports.fromElmToJS.subscribe(function(world) {
                console.log(world)
            });
            app.ports.fromJSToElm.send('hello');
```

```
        </script>
    </body>
</html>
```

REPL

A good way to learn about Elm is to try writing some expressions in the REPL (Read-Eval-Print Loop). Open a console in your `elm-app` folder (that you have created in the Initialize and build phase) and try the following:

```
$ elm repl
---- elm-repl 0.17.1 -----------------------------------------------------------
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
> 2 + 2
4 : number
> \x -> x
<function> : a -> a
> (\x -> x + x)
<function> : number -> number
> (\x -> x + x) 2
4 : number
>
```

`elm-repl` is actually a pretty powerful tool. Let's say you create a `Test.elm` file inside your `elm-app` folder with the following code:

```
module Test exposing (..)


a = 1


b = "Hello"
```

Now, you go back to your REPL (which has stayed opened) and type:

```
import Test exposing (..)
> a
1 : number
> b
"Hello" : String
>
```

Even more impressive, if you add a new definition to your `Test.elm` file, such as

```
s = """
Hello,
Goodbye.
"""
```

Save your file, go back once again to your REPL, and without importing `Test` again, the new definition is available immediately:

```
> s
"\nHello,\nGoodbye.\n" : String
>
```

It's really convenient when you want to write expressions which span many lines. It's also very useful to quickly test functions that you have just defined. Add the following to your file:

```
f x =
  x + x * x
```

Save and go back to the REPL:

```
> f
<function> : number -> number
> f 2
6 : number
> f 4
20 : number
>
```

Each time you modify and save a file that you have imported, and you go back to the REPL and try to do anything, the full file is recompiled. Therefore it will tell you about any error in your code. Add this:

```
c = 2 ++ 2
```

Try that:

```
> 0
-- TYPE MISMATCH -------------------------------------------------- ././Test.elm

The left argument of (++) is causing a type mismatch.

22|     2 ++ 2
        ^
(++) is expecting the left argument to be a:

    appendable

But the left argument is:

    number

Hint: Only strings, text, and lists are appendable.


>
```

To conclude this introduction to the REPL, let's add that `elm-repl` also knows about the packages that you have installed with `elm package install`. For instance:

```
> import Html.App
> Html.App.beginnerProgram
```

```
<function>
    : { model : a, update : b -> a -> a, view : a -> Html.Html b }
      -> Platform.Program Basics.Never
>
```

Local Build Server (Elm Reactor)

Elm Reactor is the essential tool for prototyping your application.

Please note, that you will not be able to compile `Main.elm` with Elm Reactor, if you are using Http.App.programWithFlags or Ports

Running elm-reactor in a projects directory will start a web server with a project explorer, that allows you to compile every separate component.

Any changes you make to your code are updated when you reload the page.

```
$ elm-reactor                     # launch elm-reactor on localhost:8000
$ elm-reactor -a=0.0.0.0 -p=3000  # launch elm-reactor on 0.0.0.0:3000
```

Read Getting started with Elm Language online: https://riptutorial.com/elm/topic/1011/getting-started-with-elm-language

Chapter 2: Backend Integration

Examples

Basic elm Http.post json request to node.js express server

Live upcase server that returns error when input string is longer than 10 characters.

Server:

```
const express = require('express'),
    jsonParser = require('body-parser').json(),
    app = express();

// Add headers to work with elm-reactor
app.use((req, res, next) => {
    res.setHeader('Access-Control-Allow-Origin', 'http://localhost:8000');
    res.setHeader('Access-Control-Allow-Methods', 'POST, OPTIONS');
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');
    res.setHeader('Access-Control-Allow-Credentials', true);
    next();
});

app.post('/upcase', jsonParser, (req, res, next) => {
    // Just an example of possible invalid data for an error message demo
    if (req.body.input && req.body.input.length < 10) {
        res.json({
            output: req.body.input.toUpperCase()
        });
    } else {
        res.status(500).json({
            error: `Bad input: '${req.body.input}'`
        });
    }
});

const server = app.listen(4000, () => {
    console.log('Server is listening at http://localhost:4000/upcase');
});
```

Client:

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Http
import Json.Decode as JD
import Json.Encode as JE

main : Program Never Model Msg
main =
    Html.program
        { init = init
        , view = view
        , update = update
```

```
        , subscriptions = subscriptions
        }

-- MODEL

type alias Model =
    { output: String
    , error: Maybe String
    }

init : (Model, Cmd Msg)
init =
    ( Model "" Nothing
    , Cmd.none
    )

-- UPDATE

type Msg
    = UpcaseRequest ( Result Http.Error String )
    | InputString String

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        UpcaseRequest (Ok response) ->
            ( { model | output = response, error = Nothing }, Cmd.none )

        UpcaseRequest (Err err) ->
            let
                errMsg = case err of
                    Http.Timeout ->
                        "Request timeout"

                    Http.NetworkError ->
                        "Network error"

                    Http.BadPayload msg _ ->
                        msg

                    Http.BadStatus response ->
                        case JD.decodeString upcaseErrorDecoder response.body of
                            Ok errStr ->
                                errStr

                            Err _ ->
                                response.status.message

                    Http.BadUrl msg ->
                        "Bad url: " ++ msg
            in
                ( { model | output = "", error = Just errMsg }, Cmd.none )

        InputString str ->
            ( model, upcaseRequest str )

-- VIEW

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

```
        outDiv = case model.error of
            Nothing ->
                div []
                    [ label [ for "outputUpcase" ] [ text "Output" ]
                    , input [ type_ "text", id "outputUpcase", readonly True, value
model.output ] []
                    ]

            Just err ->
                div []
                    [ label [ for "errorUpcase" ] [ text "Error" ]
                    , input [ type_ "text", id "errorUpcase", readonly True, value err ] []
                    ]
    in
        div []
            [ div []
                [ label [ for "inputToUpcase" ] [ text "Input" ]
                , input [ type_ "text", id "inputToUpcase", onInput InputString ] []
                ]
            , outDiv
            ]

-- SUBSCRIPTIONS

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

-- HELPERS

upcaseSuccessDecoder : JD.Decoder String
upcaseSuccessDecoder = JD.field "output" JD.string

upcaseErrorDecoder : JD.Decoder String
upcaseErrorDecoder = JD.field "error" JD.string

upcaseRequestEncoder : String -> JE.Value
upcaseRequestEncoder str = JE.object [ ( "input", JE.string str ) ]

upcaseRequest : String -> Cmd Msg
upcaseRequest str =
    let
        req = Http.post "http://localhost:4000/upcase" ( Http.jsonBody <| upcaseRequestEncoder
str ) upcaseSuccessDecoder
    in
        Http.send UpcaseRequest req
```

Read Backend Integration online: https://riptutorial.com/elm/topic/8087/backend-integration

Chapter 3: Collecting Data: Tuples, Records and Dictionaries

Examples

Tuples

Tuples are ordered lists of values of any type.

```
(True, "Hello!", 42)
```

It is impossible to change the structure of a Tuple or update the value.

Tuples in Elm are considered a primitive data type, which means that you don't need to import any modules to use Tuples.

Accessing values

Basics module has two helper functions for accessing values of a Tuple with a length of two `( a, b )` without using pattern matching:

```
fst (True, "Hello!") -- True
snd (True, "Hello!") -- "Hello!"
```

Access values of tuples with a bigger length is done through pattern matching.

Pattern matching

Tuples are extremely useful in combination with pattern matching:

```
toggleFlag: (Sting, Bool) -> (Sting, Bool)
toggleFlag (name, flag) =
    (name, not flag)
```

Remarks on Tuples

Tuples contain less than 7 values of `comparable` data type

Dictionaries

Dictionaries are implemented in a Dict core library.

> A dictionary mapping unique keys to values. The keys can be any comparable type. This includes Int, Float, Time, Char, String, and tuples or lists of comparable types.
>
> Insert, remove, and query operations all take O(log n) time.

Unlike Tuples and Records, Dictionaries can change their structure, in other words it is possible to add and remove keys.

It is possible to update a value by a key.

It is also possible to access or update a value using dynamic keys.

Accessing values

You can retrieve a value from a Dictionary by using a `Dict.get` function.

Type definition of `Dict.get`:

```
get : comparable -> Dict comparable v -> Maybe v
```

It will always return `Maybe v`, because it is possible to try to get a value by an non-existent key.

```
import Dict

initialUsers =
      Dict.fromList [ (1, "John"), (2, "Brad") ]

getUserName id =
  initialUsers
  |> Dict.get id
  |> Maybe.withDefault "Anonymous"

getUserName 2 -- "Brad"
getUserName 0 -- "Anonymous"
```

Updating values

Update operation on a Dictionary is performed by using `Maybe.map`, since the requested key might be absent.

```
import Dict

initialUsers =
  Dict.fromList [ (1, "John"), (2, "Brad") ]

updatedUsers =
  Dict.update 1 (Maybe.map (\name -> name ++ " Johnson")) initialUsers

Maybe.withDefault "No user" (Dict.get 1 updatedUsers) -- "John Johnson"
```

Records

Record is a set of key-value pairs.

```
greeter =
    { isMorning: True
    , greeting: "Good morning!"
    }
```

It is impossible to access a value by an non-existent key.

It is impossible to dynamically modify Record's structure.

Records only allow you to update values by constant keys.

Accessing values

Values can not be accessed using a dynamic key to prevent possible run-time errors:

```
isMorningKeyName =
    "isMorning "


greeter[isMorningKeyName] -- Compiler error
greeter.isMorning -- True
```

The alternative syntax for accessing the value allows you to extract the value, while piping through the Record:

```
greeter
    |> .greeting
    |> (++) " Have a nice day!" -- "Good morning! Have a nice day!"
```

Extending Types

Sometimes you'd want the signature of a parameter to constrain the record types you pass into functions. Extending record types makes the idea of supertypes unnecessary. The following example shows how this concept can be implemented:

```
type alias Person =
    { name : String
    }


type alias Animal =
    { name : String
    }
```

```
peter : Person
peter =
    { name = "Peter" }


dog : Animal
dog =
    { name = "Dog" }


getName : { a | name : String } -> String
getName livingThing =
    livingThing.name


bothNames : String
bothNames =
    getName peter ++ " " ++ getName dog
```

We could even take extending records a step further and do something like:

```
type alias Named a = { a | name : String }
type alias Totalled a = { a | total : Int }


totallyNamed : Named ( Totalled { age : Int })
totallyNamed =
  { name = "Peter Pan"
  , total = 1337
  , age = 14
  }
```

We now have ways to pass these partial types around in functions:

```
changeName : Named a -> String -> Named a
changeName a newName =
  { a | name = newName }

cptHook = changeName totallyNamed "Cpt. Hook" |> Debug.log "who?"
```

Updating values

Elm has a special syntax for Record updates:

```
model =
    { id: 1
    , score: 0
    , name: "John Doe"
    }


update model =
    { model
       | score = model.score + 100
       | name = "John Johnson"
```

```
}
```

Read Collecting Data: Tuples, Records and Dictionaries online:
https://riptutorial.com/elm/topic/2166/collecting-data--tuples--records-and-dictionaries

Chapter 4: Custom JSON Decoders

Introduction

How to use Json.Decode to create custom decoders, for example decoding into union types and user defined data types

Examples

Decoding into union type

```
import Json.Decode as JD
import Json.Decode.Pipeline as JP

type PostType = Image | Video

type alias Post = {
    id: Int
    , postType: PostType
}
-- assuming server will send int value of 0 for Image or 1 for Video
decodePostType: JD.Decoder PostType
decodePostType =
    JD.int |> JD.andThen (\postTypeInt ->
        case postTypeInt of
            0 ->
                JD.succeed Image


            1 ->
                JD.succed Video

            _ ->
                JD.fail "invalid posttype"

    )

decodePostMap : JD.Decoder Post
decodePostMap =
    JD.map2 Post
        (JD.field "id" JD.int)
        (JD.field "postType" decodePostType)

decodePostPipline : JD.Decoder Post
decodePostPipline =
    JP.decode Post
        |> JP.required "id" JD.int
        |> JP.required "postType" decodePostType
```

Read Custom JSON Decoders online: https://riptutorial.com/elm/topic/9927/custom-json-decoders

Chapter 5: Debugging

Syntax

- Debug.log "tag" anyValue

Remarks

`Debug.log` takes two parameters, a `String` to tag the debug output in the console (so you know where it's coming from / what the message corresponds to), and a value of any type. `Debug.log` executes the side-effect of logging the tag and the value to the JavaScript console, and then returns the value. The implementation in JS might look something like:

```
function log (tag, value){
    console.log(tag, value);
    return value
}
```

JavaScript has implicit conversions, so `value` doesn't have to be explicitly converted to a `String` for the above code to work. However, Elm types must be explicitly converted to a `String`, and the Native code for `Debug.log` shows this in action.

Examples

Logging a value without interrupting computations

`Debug.log`'s second argument is always returned, so you could write code like the following and it would *just work*:

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case Debug.log "The Message" msg of
        Something ->
            ...
```

Replacing `case msg of` with `case Debug.log "The Message" msg of` will cause the current message to be logged the console every time the update function is called, but changes nothing else.

Piping a Debug.log

At run time the following would display a list of url in your console and continue computation

```
payload =
    [{url:..., title:...}, {url=..., title=...}]

main =
```

```
    payload
        |> List.map .url -- only takes the url
        |> Debug.log " My list of URLs" -- pass the url list to Debug.log and return it
        |> doSomething -- do something with the url list
```

Time-traveling debugger

0.170.18.0

At the time of writing (July 2016) elm-reactor has been temporarily stripped of its time traveling functionality. It's possible to get it, though, using the `jinjor/elm-time-travel` package.

It's usage mirrors `Html.App` or `Navigation` modules' `program*` functions, for example instead of:

```
import Html.App

main =
    Html.App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

you'd write:

```
import TimeTravel.Html.App

main =
    TimeTravel.Html.App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

(Of course, after installing the package with `elm-package`.)

The interface of your app changes as a result, see one of the demos.

0.18.0

Since version **0.18.0** you can simply can compile your program with the `--debug` flag and get time travel debugging with no additional effort.

Debug.Crash

```
case thing of
    Cat ->
        meow
    Bike ->
        ride
    Sandwich ->
```

```
        eat
    _ ->
        Debug.crash "Not yet implemented"
```

You can use `Debug.crash` when you want the program to fail, typically used when you're in the middle of implementing a `case` expression. It is *not* recommended to use `Debug.crash` instead of using a `Maybe` or `Result` type for unexpected inputs, but typically only during the course of development (i.e. you typically wouldn't publish Elm code which uses `Debug.crash`).

`Debug.crash` takes one `String` value, the error message to show when crashing. Note that Elm will also output the name of the module and the line of the crash, and if the crash is in a `case` expression, it will indicate the value of the `case`.

Read Debugging online: https://riptutorial.com/elm/topic/2845/debugging

Chapter 6: Functions and Partial Application

Syntax

- -- defining a function with no arguments looks the same as simply defining a value
  language = "Elm"
- -- calling a function with no arguments by stating its name
  language
- -- parameters are separated by spaces and follow the function's name
  add x y = x + y
- -- call a function in the same way
  add 5 2
- -- partially apply a function by providing only some of its parameters
  increment = add 1
- -- use the |> operator to pass the expression on the left to the function on the right
  ten = 9 |> increment
- -- the <| operator passes the expression on the right to the function on the left
  increment <| add 5 4
- -- chain/compose two functions together with the >> operator
  backwardsYell = String.reverse >> String.toUpper
- -- the << works the same in the reverse direction
  backwardsYell = String.toUpper << String.reverse
- -- a function with a non-alphanumeric name in parentheses creates a new operator
  (#) x y = x * y
  ten = 5 # 2
- -- any infix operator becomes a normal function when you wrap it in parentheses
  ten = (+) 5 5
- -- optional type annotations appear above function declarations
  isTen : Int -> Bool
  isTen n = if n == 10 then True else False

Examples

Overview

Function application syntax in Elm does not use parenthesis or commas, and is instead whitespace-sensitive.

To define a function, specify its name `multiplyByTwo` and arguments `x`, any operations after equal sign `=` is what returned from a function.

```
multiplyByTwo x =
    x * 2
```

To call a function, specify its name and arguments:

```
multiplyByTwo 2  -- 4
```

Note that syntax like `multiplyByTwo(2)` is not necessary (even though the compiler doesn't complain). The parentheses only serve to resolve precedence:

```
> multiplyByTwo multiplyByTwo 2
-- error, thinks it's getting two arguments, but it only needs one

> multiplyByTwo (multiplyByTwo 2)
4 : number

> multiplyByTwo 2 + 2
6 : number
-- same as (multiplyByTwo 2) + 2

> multiplyByTwo (2 + 2)
8 : number
```

Lambda expressions

Elm has a special syntax for lambda expressions or anonymous functions:

```
\arguments -> returnedValue
```

For example, as seen in `List.filter`:

```
> List.filter (\num -> num > 1) [1,2,3]
[2,3] : List number
```

More to the depth, a backward slash, `\`, is used to mark the beginning of lambda expression, and the arrow, `->`, is used to delimit arguments from the function body. If there are more arguments, they get separated by a space:

```
normalFunction x y = x + y
-- is equivalent to
lambdaFunction = \x y -> x + y

> normalFunction 1 2
3 : number

> lambdaFunction 1 2
3 : number
```

Local variables

It is possible to define local variables inside a function to

- reduce code repetition
- give name to subexpressions
- reduce the amount of passed arguments.

The construct for this is `let ... in ...`.

```
bigNumbers =
    let
        allNumbers =
            [1..100]

        isBig number =
            number > 95
    in
        List.filter isBig allNumbers

> bigNumbers
[96,97,98,99,100] : List number

> allNumbers
-- error, doesn't know what allNumbers is!
```

The order of definitions in the first part of `let` doesn't matter!

```
outOfOrder =
    let
        x =
            y + 1  -- the compiler can handle this

        y =
            100
    in
        x + y

> outOfOrder
201 : number
```

Partial Application

Partial application means calling a function with less arguments than it has and saving the result as another function (that waits for the rest of the arguments).

```
multiplyBy: Int -> Int -> Int
multiplyBy x y =
    x * y


multiplyByTwo : Int -> Int  -- one Int has disappeared! we now know what x is.
multiplyByTwo =
    multiplyBy 2


> multiplyByTwo 2
4 : Int

> multiplyByTwo 4
8 : Int
```

As an academic sidenote, Elm can do this because of currying behind the scenes.

Strict and delayed evaluation

In elm, a function's value is computed when the last argument is applied. In the example below, the diagnostic from `log` will be printed when f is invoked with 3 arguments or a curried form of f is applied with the last argument.

```
import String
import Debug exposing (log)

f a b c = String.join "," (log "Diagnostic" [a,b,c]) -- <function> : String -> String ->
String -> String

f2 = f "a1" "b2" -- <function> : String -> String

f "A" "B" "C"
-- Diagnostic: ["A","B","C"]
"A,B,C" : String

f2 "c3"
-- Diagnostic: ["a1","b2","c3"]
"a1,b2,c3" : String
```

At times you'll want to prevent a function from being applied right away. A typical use in elm is `Lazy.lazy` which provides an abstraction for controlling when functions are applied.

```
lazy : (() -> a) -> Lazy a
```

Lazy computations take a function of one `()` or `Unit` type argument. The unit type is conventionally the type of a placeholder argument. In an argument list, the corresponding argument is specified as _, indicating that the value isn't used. The unit value in elm is specified by the special symbol `()` which can conceptually represent an empty tuple, or a hole. It resembles the empty argument list in C, Javascript and other languages that use parenthesis for function calls, but it's an ordinary value.

In our example, `f` can be protected from being evaluated immediately with a lambda:

```
doit f = f () -- <function> : (() -> a) -> a
whatToDo = \_ -> f "a" "b" "c" -- <function> : a -> String
-- f is not evaluated yet

doit whatToDo
-- Diagnostic: ["a","b","c"]
"a,b,c" : String
```

Function evaluation is delayed any time a function is partially applied.

```
defer a f = \_ -> f a -- <function> : a -> (a -> b) -> c -> b

delayF = f "a" "b" |> defer "c" -- <function> : a -> String

doit delayF
-- Diagnostic: ["a","b","c"]
"a,b,c" : String
```

Elm has an `always` function, which cannot be used to delay evaluation. Because elm evaluates all function arguments regardless of whether and when the result of the function application is used, wrapping a function application in `always` won't cause a delay, because `f` is fully applied as a parameter to `always`.

```
alwaysF = always (f "a" "b" "c") -- <function> : a -> String
-- Diagnostic: ["a","b","c"] -- Evaluation wasn't delayed.
```

Infix operators and infix notation

Elm allows the definition of custom infix operators.

Infix operators are defined using parenthesis around the name of a function.

Consider this example of infix operator for construction Tuples `1 => True -- (1, True)`:

```
(=>) : a -> b -> ( a, b )
(=>) a b =
    ( a, b )
```

Most of the functions in Elm are defined in prefix notation.

Apply any function using infix notation by specifying the first argument before the function name enclosed with grave accent character:

```
import List exposing (append)


append [1,1,2] [3,5,8]   -- [1,1,2,3,5,8]
[1,1,2] `append` [3,5,8] -- [1,1,2,3,5,8]
```

Read Functions and Partial Application online: https://riptutorial.com/elm/topic/2051/functions-and-partial-application

Chapter 7: Json.Decode

Remarks

`Json.Decode` exposes two functions to decode a payload, first one is `decodeValue` which tries to decode a `Json.Encode.Value`, the second one is `decodeString` which tries to decode a JSON string. Both function take 2 parameters, a decoder and a `Json.Encode.Value` or Json string.

Examples

Decoding a list

The following example can be tested on https://ellie-app.com/m9tk39VpQg/0.

```
import Html exposing (..)
import Json.Decode

payload =
  """
  ["fu", "bar"]
  """

main =
  Json.Decode.decodeString decoder payload -- Ok ["fu","bar"]
  |> toString
  |> text

decoder =
  Json.Decode.list Json.Decode.string
```

Pre-decode a field and decode the rest depending on that decoded value

The following examples can be tested on https://ellie-app.com/m9vmQ8NcMc/0.

```
import Html exposing (..)
import Json.Decode

payload =
  """
  [ { "bark": true, "tag": "dog", "name": "Zap", "playful": true }
  , { "whiskers": true, "tag" : "cat", "name": "Felix" }
  , {"color": "red", "tag": "tomato"}
  ]
  """

-- OUR MODELS

type alias Dog =
  { bark: Bool
  , name: String
  , playful: Bool
  }
```

```
type alias Cat =
  { whiskers: Bool
  , name: String
  }

-- OUR DIFFERENT ANIMALS

type Animal
  = DogAnimal Dog
  | CatAnimal Cat
  | NoAnimal

main =
  Json.Decode.decodeString decoder payload
  |> toString
  |> text

decoder =
  Json.Decode.field "tag" Json.Decode.string
    |> Json.Decode.andThen animalType
    |> Json.Decode.list

animalType tag =
  case tag of
    "dog" ->
      Json.Decode.map3 Dog
          (Json.Decode.field "bark" Json.Decode.bool)
          (Json.Decode.field "name" Json.Decode.string)
          (Json.Decode.field "playful" Json.Decode.bool)
        |> Json.Decode.map DogAnimal
    "cat" ->
      Json.Decode.map2 Cat
          (Json.Decode.field "whiskers" Json.Decode.bool)
          (Json.Decode.field "name" Json.Decode.string)
        |> Json.Decode.map CatAnimal
    _ ->
      Json.Decode.succeed NoAnimal
```

Decoding JSON from Rust enum

This is useful if you use rust in the backend and elm on the front end

```
enum Complex{
    Message(String),
    Size(u64)
}

let c1 = Complex::Message("hi");
let c2 = Complex::Size(1024u64);
```

The encoded Json from rust will be:

```
c1:
    {"variant": "Message",
     "fields": ["hi"]
    }
c2:
```

```
    {"variant": "Size",
     "fields": [1024]
    }
```

The decoder in elm

```
import Json.Decode as Decode exposing (Decoder)

type Complex = Message String
    | Size Int

-- decodes json to Complex type
complexDecoder: Decoder Value
complexDecoder =
    ("variant" := Decode.string `Decode.andThen` variantDecoder)

variantDecoder: String -> Decoder Value
variantDecoder variant =
    case variant of
        "Message" ->
            Decode.map Message
                ("fields" := Decode.tuple1 (\a -> a) Decode.string)
        "Size" ->
            Decode.map Size
                ("fields" := Decode.tuple1 (\a -> a) Decode.int)
        _ ->
            Debug.crash "This can't happen"
```

Usage: the data is requested from http rest api and the decoding of the payload will be

```
    Http.fromJson complexDecoder payload
```

Decoding from string will be

```
    Decode.decodeString complexDecoder payload
```

Decoding a list of records

The following code can be found in a demo here: https://ellie-app.com/mbFwJT9jD3/0

```
import Html exposing (..)
import Json.Decode exposing (Decoder)

payload =
  """
  [{
      "id": 0,
      "name": "Adam Carter",
      "work": "Unilogic",
      "email": "adam.carter@unilogic.com",
      "dob": "24/11/1978",
      "address": "83 Warner Street",
      "city": "Boston",
      "optedin": true
    },
```

```
    {
      "id": 1,
      "name": "Leanne Brier",
      "work": "Connic",
      "email": "leanne.brier@connic.org",
      "dob": "13/05/1987",
      "address": "9 Coleman Avenue",
      "city": "Toronto",
      "optedin": false
    }]
  """

type alias User =
  { name: String
  , work: String
  , email: String
  , dob: String
  , address: String
  , city: String
  , optedin: Bool
  }

main =
  Json.Decode.decodeString decoder payload
  |> toString
  |> text

decoder: Decoder (List User)
decoder =
    Json.Decode.map7 User
    (Json.Decode.field "name" Json.Decode.string)
    (Json.Decode.field "work" Json.Decode.string)
    (Json.Decode.field "email" Json.Decode.string)
    (Json.Decode.field "dob" Json.Decode.string)
    (Json.Decode.field "address" Json.Decode.string)
    (Json.Decode.field "city" Json.Decode.string)
    (Json.Decode.field "optedin" Json.Decode.bool)
    |> Json.Decode.list
```

Decode a Date

In case you have json with an ISO date string like this

```
JSON.stringify({date: new Date()})
// -> "{"date":"2016-12-12T13:24:34.470Z"}"
```

You can map it to elm `Date` type:

```
import Html exposing (text)
import Json.Decode as JD
import Date

payload = """{"date":"2016-12-12T13:24:34.470Z"}"""

dateDecoder : JD.Decoder Date.Date
dateDecoder =
  JD.string
    |> JD.andThen ( \str ->
```

```
            case Date.fromString str of
              Err err -> JD.fail err
              Ok date -> JD.succeed date )

payloadDecoder : JD.Decoder Date.Date
payloadDecoder =
  JD.field "date" dateDecoder

main =
  JD.decodeString payloadDecoder payload
  |> toString
  |> text
```

Decode a List of Objects Containing Lists of Objects

See Ellie for a working example. This example uses the NoRedInk/elm-decode-pipeline module.

Given a list of JSON objects, which themselves contain lists of JSON objects:

```
[
  {
    "id": 0,
    "name": "Item 1",
    "transactions": [
      { "id": 0, "amount": 75.00 },
      { "id": 1, "amount": 25.00 }
    ]
  },
  {
    "id": 1,
    "name": "Item 2",
    "transactions": [
      { "id": 0, "amount": 50.00 },
      { "id": 1, "amount": 15.00 }
    ]
  }
]
```

If the above string is in the `payload` string, that can be decoded using the following:

```
module Main exposing (main)

import Html exposing (..)
import Json.Decode as Decode exposing (Decoder)
import Json.Decode.Pipeline as JP
import String


type alias Item =
    { id : Int
    , name : String
    , transactions : List Transaction
    }


type alias Transaction =
    { id : Int
```

```
    , amount : Float
    }


main =
    Decode.decodeString (Decode.list itemDecoder) payload
        |> toString
        |> String.append "JSON "
        |> text



itemDecoder : Decoder Item
itemDecoder =
    JP.decode Item
        |> JP.required "id" Decode.int
        |> JP.required "name" Decode.string
        |> JP.required "transactions" (Decode.list transactionDecoder)


transactionDecoder : Decoder Transaction
transactionDecoder =
    JP.decode Transaction
        |> JP.required "id" Decode.int
        |> JP.required "amount" Decode.float
```

Read Json.Decode online: https://riptutorial.com/elm/topic/2849/json-decode

Chapter 8: Lists and Iteration

Remarks

The `List` (linked list) shines in **sequential access**:

- accessing the first element
- prepending to the front of the list
- deleting from the front of the list

On the other hand, it's not ideal for **random access** (ie. getting nth element) and **traversation in reverse order**, and you might have better luck (and performance) with the `Array` data structure.

Examples

Creating a list by range

0.18.0

Prior to **0.18.0** you can create ranges like this:

```
> range = [1..5]
[1,2,3,4,5] : List number
>
> negative = [-5..3]
[-5,-4,-3,-2,-1,0,1,2,3] : List number
```

0.18.0

In **0.18.0** The `[1..5]` syntax has been removed.

```
> range = List.range 1 5
[1,2,3,4,5] : List number
>
> negative = List.range -5 3
[-5,-4,-3,-2,-1,0,1,2,3] : List number
```

Ranges created by this syntax are always **inclusive** and the **step** is always **1**.

Creating a list

```
> listOfNumbers = [1,4,99]
[1,4,99] : List number
>
> listOfStrings = ["Hello","World"]
["Hello","World"] : List String
>
> emptyList = []   -- can be anything, we don't know yet
[] : List a
```

```
>
```

Under the hood, `List` (linked list) is constructed by the `::` function (called "cons"), which takes two arguments: an element, known as the head, and a (possibly empty) list the head is prepended to.

```
> withoutSyntaxSugar = 1 :: []
[1] : List number
>
> longerOne = 1 :: 2 :: 3 :: []
[1,2,3] : List number
>
> nonemptyTail = 1 :: [2]
[1,2] : List number
>
```

`List` can only take values of one type, so something like `[1,"abc"]` is not possible. If you need this, use tuples.

```
> notAllowed = [1,"abc"]
==================================== ERRORS ====================================

-- TYPE MISMATCH --------------------------------------------- repl-temp-000.elm

The 1st and 2nd elements are different types of values.

8|              [1,"abc"]
               ^^^^^
The 1st element has this type:

    number

But the 2nd is:

    String

Hint: All elements should be the same type of value so that we can iterate
through the list without running into unexpected values.


>
```

Getting elements

```
> ourList = [1,2,3,4,5]
[1,2,3,4,5] : List number
>
> firstElement = List.head ourList
Just 1 : Maybe Int
>
> allButFirst = List.tail ourList
Just [2,3,4,5] : Maybe (List Int)
```

This wrapping into `Maybe` type happens because of the following scenario:

What should `List.head` return for an empty list? (Remember, Elm doesn't have exceptions or nulls.)

```
> headOfEmpty = List.head []
Nothing : Maybe Int
>
> tailOfEmpty = List.tail []
Nothing : Maybe (List Int)
>
> tailOfAlmostEmpty = List.tail [1] -- warning ... List is a *linked list* :)
Just [] : Maybe (List Int)
```

Transforming every element of a list

`List.map : (a -> b) -> List a -> List b` is a higher-order function that applies a one-parameter function to each element of a list, returning a new list with the modified values.

```
import String

ourList : List String
ourList =
    ["wubba", "lubba", "dub", "dub"]

lengths : List Int
lengths =
    List.map String.length ourList
-- [5,5,3,3]

slices : List String
slices =
    List.map (String.slice 1 3) ourList
-- ["ub", "ub", "ub", "ub"]
```

If you need to know the index of the elements you can use `List.indexedMap : (Int -> a -> b) -> List a -> List b`:

```
newList : List String
newList =
    List.indexedMap (\index element -> String.concat [toString index, ": ", element]) ourList
-- ["0: wubba","1: lubba","2: dub","3: dub"]
```

Filtering a list

`List.filter : (a -> Bool) -> List a -> List a` is a higher-order function which takes a one-parameter function from any value to a boolean, and applies that function to every element of a given list, keeping only those elements for which the function returns `True` on. The function that `List.filter` takes as its first parameter is often referred to as a predicate.

```
import String

catStory : List String
catStory =
    ["a", "crazy", "cat", "walked", "into", "a", "bar"]
```

```
-- Any word with more than 3 characters is so long!
isLongWord : String -> Bool
isLongWord string =
    String.length string > 3

longWordsFromCatStory : List String
longWordsFromCatStory =
    List.filter isLongWord catStory
```

Evaluate this in `elm-repl`:

```
> longWordsFromCatStory
["crazy", "walked", "into"] : List String
>
> List.filter (String.startsWith "w") longWordsFromCatStory
["walked"] : List String
```

Pattern Matching on a list

We can match on lists like any other data type, though they are somewhat unique, in that the constructor for building up lists is the infix function `::`. (See the example Creating a list for more on how that works.)

```
matchMyList : List SomeType -> SomeOtherType
matchMyList myList =
    case myList of
        [] ->
            emptyCase

        (theHead :: theRest) ->
            doSomethingWith theHead theRest
```

We can match as many elements in the list as we want:

```
hasAtLeast2Elems : List a -> Bool
hasAtLeast2Elems myList =
    case myList of
        (e1 :: e2 :: rest) ->
            True

        _ ->
            False

hasAtLeast3Elems : List a -> Bool
hasAtLeast3Elems myList =
    case myList of
        (e1 :: e2 :: e3 :: rest) ->
            True

        _ ->
            False
```

Getting nth element from the list

`List`

doesn't support "random access", which means it takes more work to get, say, the fifth element from the list than the first element, and as a result there's no `List.get nth list` function. One has to go all the way from the beginning (`1 -> 2 -> 3 -> 4 -> 5`).

If you need random access, you might get better results (and performance) with random access data structures, like `Array`, where taking the first element takes the same amount of work as taking, say, the 1000th. (complexity O(1)).

Nevertheless, it's possible **(but discouraged)** to get nth element:

```
get : Int -> List a -> Maybe a
get nth list =
    list
        |> List.drop (nth - 1)
        |> List.head

fifth : Maybe Int
fifth = get 5 [1..10]
--    = Just 5

nonexistent : Maybe Int
nonexistent = get 5 [1..3]
--          = Nothing
```

Again, this takes significantly more work the bigger the `nth` argument is.

Reducing a list to a single value

In Elm, reducing functions are called "folds", and there are two standard methods to "fold" values up: from the left, `foldl`, and from the right, `foldr`.

```
> List.foldl (+) 0 [1,2,3]
6 : number
```

The arguments to `foldl` and `foldr` are:

- **reducing function**: `newValue -> accumulator -> accumulator`
- **accumulator** starting value
- **list** to reduce

One more example with custom function:

```
type alias Counts =
    { odd : Int
    , even : Int
    }

addCount : Int -> Counts -> Counts
addCount num counts =
    let
        (incOdd, incEven) =
            if num `rem` 2 == 0
                then (0,1)
```

```
                  else (1,0)
     in
         { counts
             | odd = counts.odd + incOdd
             , even = counts.even + incEven
         }

> List.foldl
       addCount
       { odd = 0, even = 0 }
       [1,2,3,4,5]
{ odd = 3, even = 2 } : Counts
```

In the first example above the program goes like this:

```
List.foldl (+) 0 [1,2,3]
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

```
List.foldr (+) 0 [1,2,3]
1 + (2 + (3 + 0))
1 + (2 + 3)
1 + 5
6
```

In the case of a commutative function like `(+)` there's not really a difference.

But see what happens with `(::)`:

```
List.foldl (::) [] [1,2,3]
3 :: (2 :: (1 :: []))
3 :: (2 :: [1])
3 :: [2,1]
[3,2,1]
```

```
List.foldr (::) [] [1,2,3]
1 :: (2 :: (3 :: []))
1 :: (2 :: [3])
1 :: [2,3]
[1,2,3]
```

Creating a list by repeating a value

```
> List.repeat 3 "abc"
["abc","abc","abc"] : List String
```

You can give `List.repeat` any value:

```
> List.repeat 2 {a = 1, b = (2,True)}
[{a = 1, b = (2,True)}, {a = 1, b = (2,True)}]
  : List {a : Int, b : (Int, Bool)}
```

Sorting a list

By default, `List.sort` sorts in ascending order.

```
> List.sort [3,1,5]
[1,3,5] : List number
```

`List.sort` needs the list elements to be `comparable`. That means: `String`, `Char`, `number` (`Int` and `Float`), `List` of `comparable` or tuple of `comparable`.

```
> List.sort [(5,"ddd"),(4,"zzz"),(5,"aaa")]
[(4,"zzz"),(5,"aaa"),(5,"ddd")] : List ( number, String )

> List.sort [[3,4],[2,3],[4,5],[1,2]]
[[1,2],[2,3],[3,4],[4,5]] : List (List number)
```

You can't sort lists of `Bool` or objects with `List.sort`. For that see Sorting a list with custom comparator.

```
> List.sort [True, False]
-- error, can't compare Bools
```

Sorting a list with custom comparator

`List.sortWith` allows you to sort lists with data of any shape - you supply it with a comparison function.

```
compareBools : Bool -> Bool -> Order
compareBools a b =
    case (a,b) of
        (False, True) ->
            LT

        (True, False) ->
            GT

        _ ->
            EQ

> List.sortWith compareBools [False, True, False, True]
[False, False, True, True] : List Bool
```

Reversing a list

Note: this is not very efficient due to the nature of `List` (see Remarks below). It will be better to **construct the list the "right" way from the beginning** than to construct it and then reverse it.

```
> List.reverse [1,3,5,7,9]
[9,7,5,3,1] : List number
```

Sorting a list in descending order

By default `List.sort` sorts in ascending order, with the `compare` function.

There are two ways to sort in descending order: one efficient and one inefficient.

1. **The efficient way**: `List.sortWith` and a descending comparison function.

```
descending a b =
    case compare a b of
      LT -> GT
      EQ -> EQ
      GT -> LT

> List.sortWith descending [1,5,9,7,3]
[9,7,5,3,1] : List number
```

2. The inefficient way **(discouraged!)**: `List.sort` and then `List.reverse`.

```
> List.reverse (List.sort [1,5,9,7,3])
[9,7,5,3,1] : List number
```

Sorting a list by a derived value

`List.sortBy` allows to use a function on the elements and use its result for the comparison.

```
> List.sortBy String.length ["longest","short","medium"]
["short","medium","longest"] : List String
-- because the lengths are: [7,5,6]
```

It also nicely works with record accessors:

```
people =
    [ { name = "John", age = 43 }
    , { name = "Alice", age = 30 }
    , { name = "Rupert", age = 12 }
    ]

> List.sortBy .age people
[ { name = "Rupert", age = 12 }
, { name = "Alice", age = 30 }
, { name = "John", age = 43 }
] : List {name: String, age: number}

> List.sortBy .name people
[ { name = "Alice", age = 30 }
, { name = "John", age = 43 }
, { name = "Rupert", age = 12 }
] : List {name: String, age: number}
```

Read Lists and Iteration online: https://riptutorial.com/elm/topic/1635/lists-and-iteration

Chapter 9: Making complex update functions with ccapndave/elm-update-extra

Introduction

ccapndave/elm-update-extra is a fantastic package which helps you handle more complex updating functions, and may be very useful.

Examples

Message which call a list of messages

Using `sequence` function you can easily describe a message that calls a list of other messages. It's useful when dealing with semantics of your messages.

Example 1: You are making a game engine, and you need to refresh the screen on every frame.

```
module Video exposing (..)
type Message = module Video exposing (..)

import Update.Extra exposing (sequence)

-- Model definition [...]

type Message
    = ClearBuffer
    | DrawToBuffer
    | UpdateLogic
    | Update

update : Message -> Model -> (Model, Cmd)
update msg model =
    case msg of
        ClearBuffer ->
            -- do something
        DrawToBuffer ->
            -- do something
        UpdateLogic ->
            -- do something
        Update ->
            model ! []
                |> sequence update [ ClearBuffer
                                   , DrawToBuffer
                                   , UpdateLogic]
```

Chaining messages with andThen

The `andThen` function allows update call composition. Can be used with the pipeline operator (`|>`) to chain updates.

Example: You are making a document editor, and you want that each modification message you send to your document, you also save it:

```
import Update.Extra exposing (andThen)
import Update.Extra.Infix exposing (..)

-- type alias Model = [...]

type Message
    = ModifyDocumentWithSomeSettings
    | ModifyDocumentWithOtherSettings
    | SaveDocument

update : Model -> Message -> (Model, Cmd)
update model msg =
    case msg of
        ModifyDocumentWithSomeSettings ->
            -- make the modifications
            (modifiedModel, Cmd.none)
            |> andThen SaveDocument
        ModifyDocumentWithOtherSettings ->
            -- make other modifications
            (modifiedModel, Cmd.none)
            |> andThen SaveDocument
        SaveDocument ->
            -- save document code
```

If you import also `Update.Extra.Infix exposing (..)` you may be able to use the infix operator:

```
update : Model -> Message -> (Model, Cmd)
update model msg =
    case msg of
        ModifyDocumentWithSomeSettings ->
            -- make the modifications
            (modifiedModel, Cmd.none)
            :> andThen SaveDocument
        ModifyDocumentWithOtherSettings ->
            -- make other modifications
            (modifiedModel, Cmd.none)
            :> SaveDocument
        SaveDocument ->
            -- save document code
```

Read Making complex update functions with ccapndave/elm-update-extra online: https://riptutorial.com/elm/topic/9737/making-complex-update-functions-with-ccapndave-elm-update-extra

Chapter 10: Pattern Matching

Examples

Function arguments

```
type Dog = Dog String

dogName1 dog =
    case dog of
      Dog name ->
         name

dogName2 (Dog name) ->
     name
```

`dogName1` and `dogName2` are equivalent. Note that this only works for ADTs that have a single constructor.

```
type alias Pet =
   { name: String
   , weight: Float

   }

render : Pet -> String
render ({name, weight} as pet) =
    (findPetEmoji pet) ++ " " ++ name ++ " weighs " ++ (toString weight)

findPetEmoji : Pet -> String
findPetEmoji pet =
    Debug.crash "Implementation TBD"
```

Here we deconstruct a record and also get a reference to the undeconstructed record.

Single type deconstructed argument

```
type ProjectIdType = ProjectId String

getProject : ProjectIdType -> Cmd Msg
getProject (ProjectId id) =
    Http.get <| "/projects/" ++ id
```

Read Pattern Matching online: https://riptutorial.com/elm/topic/7168/pattern-matching

Chapter 11: Ports (JS interop)

Syntax

- Elm (receiving): port functionName : (value -> msg) -> Sub msg
- JS (sending): app.ports.functionName.send(value)
- Elm (sending): port functionName : args -> Cmd msg
- JS (receiving): app.ports.functionName.subscribe(function(args) { ... });

Remarks

Consult http://guide.elm-lang.org/interop/javascript.html from *The Elm Guide* to aid in understanding these examples.

Examples

Overview

A module, that is using Ports should have `port` keyword in it's module definition.

```
port module Main exposing (..)
```

It is impossible to use ports with `Html.App.beginnerProgram`, since it does not allow using Subscriptions or Commands.

Ports are integrated in to update loop of `Html.App.program` or `Html.App.programWithFlags`.

Note

`program` and `programWithFlags` in elm 0.18 are inside the package `Html` instead of `Html.App`.

Outgoing

Outgoing ports are used as Commands, that you return from your `update` function.

Elm side

Define outgoing port:

```
port output : () -> Cmd msg
```

In this example we send an empty Tuple, just to trigger a subscription on the JavaScript side.

To do so, we have to apply `output` function with an empty Tuple as argument, to get a Command for sending the outgoing data from Elm.

```
update msg model =
    case msg of
        TriggerOutgoing data ->
            ( model, output () )
```

JavaScript side

Initialize the application:

```
var root = document.body;
var app = Elm.Main.embed(root);
```

Subscribe to a port with a corresponding name:

```
app.ports.output.subscribe(function () {
    alert('Outgoing message from Elm!');
});
```

Note

As of `0.17.0`, immediate outgoing message to JavaScript from your `initial` state will have no effect.

```
init : ( Model, Cmd Msg )
init =
    ( Model 0, output () ) -- Nothing will happen
```

See the workaround in the example below.

Incoming

Incoming data from JavaScript is going through Subscriptions.

Elm side

First, we need to define an incoming port, using the following syntax:

```
port input : (Int -> msg) -> Sub msg
```

We can use `Sub.batch` if we have multiple subscriptions, this example will only contain one Subscription to `input port`

```
subscriptions : Model -> Sub Msg
subscriptions model =
    input Get
```

Then you have to pass the `subscriptions` to your `Html.program`:

```
main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

JavaScript side

Initialize the application:

```
var root = document.body;
var app = Elm.Main.embed(root);
```

Send the message to Elm:

```
var counter = 0;

document.body.addEventListener('click', function () {
    counter++;
    app.ports.input.send(counter);
});
```

Note

Please note, that as of `0.17.0` the immediate `app.ports.input.send(counter);` after app initialization will have no effect!

Pass all the required data for the start-up as Flags using `Html.programWithFlags`

Immediate outgoing message on start-up in 0.17.0

To send an immediate message with data to JavaScript, you have to trigger an action from your `init`.

```
init : ( Model, Cmd Msg )
init =
    ( Model 0, send SendOutgoing )


send : msg -> Cmd msg
send msg =
```

```
    Task.perform identity identity (Task.succeed msg)
```

Get started

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Trying out ports</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="elm.js"></script>
    <script>

      var node = document.getElementById('app');
      var app = Elm.Main.embed(node);

      // subscribe to messages from Elm
      app.ports.toJs.subscribe(function(messageFromElm) {
        alert(messageFromElm);
        // we could send something back by
        // app.ports.fromJs.send('Hey, got your message! Sincerely, JS');
      });

      // wait three seconds and then send a message from JS to Elm
      setTimeout(function () {
        app.ports.fromJs.send('Hello from JS');
      }, 3000);

    </script>
  </body>
</html>
```

Main.elm

```
port module Main exposing (..)

import Html

port toJs : String -> Cmd msg
port fromJs : (String -> msg) -> Sub msg

main =
   Html.program
        { init = (Nothing, Cmd.none) -- our model will be the latest message from JS (or
Nothing for 'no message yet')
        , update = update
        , view = view
        , subscriptions = subscriptions
        }

type Msg
    = GotMessageFromJs String

update msg model =
```

```
    case msg of
        GotMessageFromJs message ->
            (Just message, toJs "Hello from Elm")

view model =
    case model of
        Nothing ->
            Html.text "No message from JS yet :("
        Just message ->
            Html.text ("Last message from JS: " ++ message)

subscriptions model =
    fromJs GotMessageFromJs
```

Install the `elm-lang/html` package if you haven't yet by `elm-package install elm-lang/html --yes`.

Compile this code using `elm-make Main.elm --yes --output elm.js` so that the HTML file finds it.

If everything goes well, you should be able to open the `index.html` file with the "No message" text displayed. After three seconds the JS sends a message, Elm gets it, changes its model, sends a response, JS gets it and opens an alert.

Read Ports (JS interop) online: https://riptutorial.com/elm/topic/2200/ports--js-interop-

Chapter 12: Subscriptions

Remarks

Subscriptions are means to listen to inputs. Incoming ports, keyboard or mouse events, WebSocket messages, geolocation and page visibility changes, all can serve as inputs.

Examples

Basic subscription to Time.every event with 'unsubscribe'

0.18.0

Model is passed to subscriptions which means that every state change can modify subscriptions.

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

main : Program Never Model Msg
main =
    Html.program
        { init = init
        , update = update
        , subscriptions = subscriptions
        , view = view
        }

-- MODEL

type alias Model =
    { time: Time.Time
    , suspended: Bool
    }

init : (Model, Cmd Msg)
init =
    ( Model 0 False, Cmd.none )

-- UPDATE

type Msg
    = Tick Time.Time
    | SuspendToggle

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

        SuspendToggle ->
            ( { model | suspended = not model.suspended }, Cmd.none )
```

```
-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    if model.suspended then
        Sub.none
    else
        Time.every Time.second Tick

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ div [] [ text <| toString model ]
        , button [ onClick SuspendToggle ] [ text ( if model.suspended then "Resume" else
"Suspend" ) ]
        ]
```

Read Subscriptions online: https://riptutorial.com/elm/topic/4279/subscriptions

Chapter 13: The Elm Architecture

Introduction

The recommended way to structure your applications is dubbed 'the Elm Architecture.'

The simplest program consists of a `model` record storing all data that might be updated, a union type `Msg` that defines ways your program updates that data, a function `update` which takes the model and a `Msg` and returns a new model, and a function `view` which takes a model and returns the HTML your page will display. Anytime a function returns a `Msg`, the Elm runtime uses it to update the page.

Examples

Beginner program

Html has `beginnerProgram` mostly for learning purposes.

`beginnerProgram` is not capable of handling Subscriptions or running Commands.

It is only capable of handling user input from DOM Events.

It only requires a `view` to render the `model` and an `update` function to handle state changes.

Example

Consider this minimal example of `beginnerProgram`.

The `model` in this example consists of single `Int` value.

The `update` function has only one branch, which increments the `Int`, stored in the `model`.

The `view` renders the model and attaches click DOM Event.

See how to build the example in Initialize and build

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


main : Program Never
main =
    beginnerProgram { model = 0, view = view, update = update }


-- UPDATE
```

```
type Msg
    = Increment

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


-- VIEW


view : Int -> Html Msg
view model =
    button [ onClick Increment ] [ text ("Increment: " ++ (toString model)) ]
```

Program

`program` is a good pick, when your application does not require any external data for initialization.

It is capable of handling Subscriptions and Commands, which enables way more opportunities for handling I/O, such as HTTP communication or interop with JavaScript.

The initial state is required to return start-up Commands along with the Model.

The initialization of `program` will require `subscriptions` to be provided, along with `model`, `view` and `update`.

See the type definition:

```
program :
    { init : ( model, Cmd msg )
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    , view : model -> Html msg
    }
    -> Program Never
```

Example

The simplest way to illustrate, how you can use Subscriptions is to setup a simple Port communication with JavaScript.

See how to build the example in Initialize and build / Embedding into HTML

```
port module Main exposing (..)

import Html exposing (Html, text)
import Html exposing (program)
```

```
main : Program Never
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }


port input : (Int -> msg) -> Sub msg


-- MODEL


type alias Model =
    Int


init : ( Model, Cmd msg )
init =
    ( 0, Cmd.none )


-- UPDATE


type Msg = Incoming Int


update : Msg -> Model -> ( Model, Cmd msg )
update msg model =
    case msg of
        Incoming x ->
          ( x, Cmd.none )


-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
    input Incoming


-- VIEW


view : Model -> Html msg
view model =
    text (toString model)
```

```
<!DOCTYPE html>
<html>
    <head>
        <script src='elm.js'></script>
</head>
    <body>
    <div id='app'></div>
    <script>var app = Elm.Main.embed(document.getElementById('app'));</script>
```

```
    <button onclick='app.ports.input.send(1);'>send</button>
</body>
</html>
```

Program with Flags

`programWithFlags` has only one difference from `program`.

It can accept the data upon initialization from JavaScript:

```
var root = document.body;
var user = { id: 1, name: "Bob" };
var app = Elm.Main.embed( root, user );
```

The data, passed from JavaScript is called Flags.

In this example we are passing a JavaScript Object to Elm with user information, it is a good practice to specify a Type Alias for flags.

```
type alias Flags =
    { id: Int
    , name: String
    }
```

Flags are passed to the `init` function, producing the initial state:

```
init : Flags -> ( Model, Cmd Msg )
init flags =
    let
        { id, name } =
            flags
    in
        ( Model id name, Cmd.none )
```

You might notice the difference from it's type signature:

```
programWithFlags :
    { init : flags -> ( model, Cmd msg )          -- init now accepts flags
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    , view : model -> Html msg
    }
    -> Program flags
```

The initialization code looks almost the same, since it's only `init` function that is different.

```
main =
    programWithFlags
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

One way parent-child communication

Example demonstrates component composition and one-way message passing from parent to children.

0.18.0

Component composition relies on Message tagging with `Html.App.map`

0.18.0

In `0.18.0` `HTML.App` was collapsed into `HTML`

Component composition relies on Message tagging with `Html.map`

Example

See how to build the example in Initialise and build

```
module Main exposing (..)

import Html exposing (text, div, button, Html)
import Html.Events exposing (onClick)
import Html.App exposing (beginnerProgram)


main =
    beginnerProgram
        { view = view
        , model = init
        , update = update
        }

{- In v0.18.0 HTML.App was collapsed into HTML
   Use Html.map instead of Html.App.map
-}
view : Model -> Html Msg
view model =
    div []
        [ Html.App.map FirstCounterMsg (counterView model.firstCounter)
        , Html.App.map SecondCounterMsg (counterView model.secondCounter)
        , button [ onClick ResetAll ] [ text "Reset counters" ]
        ]


type alias Model =
    { firstCounter : CounterModel
    , secondCounter : CounterModel
    }


init : Model
init =
    { firstCounter = 0
    , secondCounter = 0
```

```
    }


type Msg
    = FirstCounterMsg CounterMsg
    | SecondCounterMsg CounterMsg
    | ResetAll


update : Msg -> Model -> Model
update msg model =
    case msg of
        FirstCounterMsg childMsg ->
            { model | firstCounter = counterUpdate childMsg model.firstCounter }

        SecondCounterMsg childMsg ->
            { model | secondCounter = counterUpdate childMsg model.secondCounter }

        ResetAll ->
            { model
                | firstCounter = counterUpdate Reset model.firstCounter
                , secondCounter = counterUpdate Reset model.secondCounter
            }


type alias CounterModel =
    Int


counterView : CounterModel -> Html CounterMsg
counterView model =
    div []
        [ button [ onClick Decrement ] [ text "-" ]
        , text (toString model)
        , button [ onClick Increment ] [ text "+" ]
        ]


type CounterMsg
    = Increment
    | Decrement
    | Reset


counterUpdate : CounterMsg -> CounterModel -> CounterModel
counterUpdate msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1

        Reset ->
            0
```

Message tagging with Html.App.map

Components define their own Messages, sent after emitted DOM Events, eg. `CounterMsg` from

Parent-child communication

```
type CounterMsg
    = Increment
    | Decrement
    | Reset
```

The view of this component will send messages of `CounterMsg` type, therefore the view type signature is `Html CounterMsg`.

To be able to reuse `counterView` inside parent component's view, we need to pass every `CounterMsg` message through parent's `Msg`.

This technique is called ***message tagging***.

Parent component must define messages for passing child messages:

```
type Msg
    = FirstCounterMsg CounterMsg
    | SecondCounterMsg CounterMsg
    | ResetAll
```

`FirstCounterMsg Increment` is a tagged message.

0.18.0

To get a `counterView` to send tagged messages, we must use the `Html.App.map` function:

```
Html.map FirstCounterMsg (counterView model.firstCounter)
```

0.18.0

The `HTML.App` package was collapsed into the `HTML` package in `v0.18.0`

To get a `counterView` to send tagged messages, we must use the `Html.map` function:

```
Html.map FirstCounterMsg (counterView model.firstCounter)
```

That changes the type signature `Html CounterMsg -> Html Msg` so it's possible to use the counter inside the parent view and handle state updates with parent's update function.

Read The Elm Architecture online: https://riptutorial.com/elm/topic/3771/the-elm-architecture

Chapter 14: Types, Type Variables, and Type Constructors

Remarks

Please play with these concepts yourself to really master them! The `elm-repl` (see the Introduction to the REPL) is probably a good place to play around with the code above. You can also play with `elm-repl` online.

Examples

Comparable data types

Comparable types are primitive types that can be compared using comparison operators from Basics module, like: `(<)`, `(>)`, `(<=)`, `(>=)`, `max`, `min`, `compare`

Comparable types in Elm are `Int`, `Float`, `Time`, `Char`, `String`, and tuples or lists of comparable types.

In documentation or type definitions they are referred as a special type variable `comparable`, eg. see type definition for `Basics.max` function:

```
max : comparable -> comparable -> comparable
```

Type Signatures

In Elm, values are declared by writing a name, an equals sign, and then the actual value:

```
someValue = 42
```

Functions are also values, with the addition of taking a value or values as arguments. They are usually written as follows:

```
double n = n * 2
```

Every value in Elm has a type. The types of the values above will be *inferred* by the compiler depending on how they are used. But it is best-practice to always explicitly declare the type of any top-level value, and to do so you write a *type signature* as follows:

```
someValue : Int
someValue =
    42

someOtherValue : Float
someOtherValue =
    42
```

As we can see, `42` can be defined as *either* an `Int` or a `Float`. This makes intuitive sense, but see **Type Variables** for more information.

Type signatures are particularly valuable when used with functions. Here's the doubling function from before:

```
double : Int -> Int
double n =
    n * 2
```

This time, the signature has a `->`, an arrow, and we'd pronounce the signature as "int to int", or "takes an integer and returns an integer". `->` indicates that by giving `double` an `Int` value as an argument, `double` will return an `Int`. Hence, it takes an integer to an integer:

```
> double
<function> : Int -> Int

> double 3
6 : Int
```

Basic Types

In `elm-repl`, type a piece of code to get its value and inferred type. Try the following to learn about the various types that exist:

```
> 42
42 : number

> 1.987
1.987 : Float

> 42 / 2
21 : Float

> 42 % 2
0 : Int

> 'e'
'e' : Char

> "e"
"e" : String

> "Hello Friend"
"Hello Friend" : String

> ['w', 'o', 'a', 'h']
['w', 'o', 'a', 'h'] : List Char

> ("hey", 42.42, ['n', 'o'])
("hey", 42.42, ['n', 'o']) : ( String, Float, List Char )

> (1, 2.1, 3, 4.3, 'c')
(1,2.1,3,4.3,'c') : ( number, Float, number', Float, Char )
```

```
> {}
{} : {}

> { hey = "Hi", someNumber = 43 }
{ hey = "Hi", someNumber = 43 } : { hey : String, someNumber : number }

> ()
() : ()
```

`{}` is the empty Record type, and `()` is the empty Tuple type. The latter is often used for the purposes of lazy evaluation. See the corresponding example in Functions and Partial Application.

Note how `number` appears uncapitalized. This indicates that it is a **Type Variable**, and moreover the particular word `number` refers to a **Special Type Variable** that can either be an `Int` or a `Float` (see the corresponding sections for more). Types though are always upper-case, such as `Char`, `Float`, `List String`, et cetera.

Type Variables

Type variables are uncapitalized names in type-signatures. Unlike their capitalized counterparts, such as `Int` and `String`, they do not represent a single type, but rather, any type. They are used to write generic functions that can operate on *any* type or types, and are particularly useful for writing operations over containers like `List` or `Dict`. The `List.reverse` function, for example, has the following signature:

```
reverse : List a -> List a
```

Which means it can work on a list of *any type value*, so `List Int`, `List (List String)`, both of those and any others can be `reversed` all the same. Hence, `a` is a type variable that can stand in for any type.

The `reverse` function could have used *any* uncapitalized variable name in its type signature, except for the handful of **special type variable** names, such as `number` (see the corresponding example on that for more information):

```
reverse : List lol -> List lol

reverse : List wakaFlaka -> List wakaFlaka
```

The names of type variables become meaningful only when there when there are *different* type variables within a single signature, exemplified by the `map` function on lists:

```
map : (a -> b) -> List a -> List b
```

`map` takes some function from any type `a` to any type `b`, along with a list with elements of some type `a`, and returns a list of elements of some type `b`, which it gets by applying the given function to every element of the list.

Let's make the signature concrete to better see this:

```
plusOne : Int -> Int
plusOne x =
    x + 1

> List.map plusOne
<function> : List Int -> List Int
```

As we can see, both `a = Int` and `b = Int` in this case. But, if `map` had a type signature like `map : (a -> a) -> List a -> List a`, then it would *only* work on functions that operate on a single type, and you'd never be able to change the type of a list by using the `map` function. But since the type signature of `map` has multiple different type variables, `a` and `b`, we can use `map` to change the type of a list:

```
isOdd : Int -> Bool
isOdd x =
    x % 2 /= 0

> List.map isOdd
<function> : List Int -> List Bool
```

In this case, `a = Int` and `b = Bool`. Hence, to be able to use functions that can take and return *different* types, you must use different type variables.

Type Aliases

Sometimes we want to give a type a more descriptive name. Let's say our app has a data type representing users:

```
{ name : String, age : Int, email : String }
```

And our functions on users have type signatures along the lines of:

```
prettyPrintUser : { name : String, age : Int, email : String } -> String
```

This could become quite unwieldy with a larger record type for a user, so let's use a *type alias* to cut down on the size and give a more meaningful name to that data structure:

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


prettyPrintUser : User -> String
```

Type aliases make it much cleaner to define and use a model for an application:

```
type alias Model =
    { count : Int
    , lastEditMade : Time
```

```
    }
```

Using `type alias` literally just aliases a type with the name you give it. Using the `Model` type above is exactly the same as using `{ count : Int, lastEditMade : Time }`. Here's an example showing how aliases are no different than the underlying types:

```
type alias Bugatti = Int

type alias Fugazi = Int

unstoppableForceImmovableObject : Bugatti -> Fugazi -> Int
unstoppableForceImmovableObject bug fug =
    bug + fug

> unstoppableForceImmovableObject 09 87
96 : Int
```

A type alias for a record type defines a constructor function with one argument for each field in declaration order.

```
type alias Point = { x : Int, y : Int }

Point 3 7
{ x = 3, y = 7 } : Point

type alias Person = { last : String, middle : String, first : String }

Person "McNameface" "M" "Namey"
{ last = "McNameface", middle = "M", first = "Namey" } : Person
```

Each record type alias has its own field order even for a compatible type.

```
type alias Person = { last : String, middle : String, first : String }
type alias Person2 = { first : String, last : String, middle : String }

Person2 "Theodore" "Roosevelt" "-"
{ first = "Theodore", last = "Roosevelt", middle = "-" } : Person2

a = [ Person "Last" "Middle" "First", Person2 "First" "Last" "Middle" ]
[{ last = "Last", middle = "Middle", first = "First" },{ first = "First", last = "Last",
middle = "Middle" }] : List Person2
```

Improving Type-Safety Using New Types

Aliasing types cuts down on boilerplate and enhances readability, but it is no more type-safe than the aliased type itself is. Consider the following:

```
type alias Email = String

type alias Name = String

someEmail = "holmes@private.com"

someName = "Benedict"
```

```
sendEmail : Email -> Cmd msg
sendEmail email = ...
```

Using the above code, we can write `sendEmail someName`, and it will compile, even though it really shouldn't, because despite names and emails both being `String`s, they are entirely different things.

We can truly distinguish one `String` from another `String` on the type-level by creating a new **type**. Here's an example that rewrites `Email` as a `type` rather than a `type alias`:

```
module Email exposing (Email, create, send)

type Email = EmailAddress String

isValid : String -> Bool
isValid email =
  -- ...validation logic

create : String -> Maybe Email
create email =
    if isValid email then
        Just (EmailAddress email)
    else
        Nothing

send : Email -> Cmd msg
send (EmailAddress email) = ...
```

Our `isValid` function does something to determine if a string is a valid email address. The `create` function checks if a given `String` is a valid email, returning a `Maybe`-wrapped `Email` to ensure that we only return validated addresses. While we can sidestep the validation check by constructing an `Email` directly by writing `EmailAddress "somestring"`, if our module declaration doesn't expose the `EmailAddress` constructor, as show here

```
module Email exposing (Email, create, send)
```

then no other module will have access to the `EmailAddress` constructor, though they can still use the `Email` type in annotations. The **only** way to build a new `Email` outside of this module is by using the `create` function it provides, and that function ensures that it will only return valid email addresses in the first place. Hence, this API automatically guides the user down the correct path via its type safety: `send` only works with values constructed by `create`, which performs a validation, and enforces handling of invalid emails since it returns a `Maybe Email`.

If you'd like to export the `Email` constructor, you could write

```
module Email exposing (Email(EmailAddress), create, send)
```

Now any file that imports `Email` can also import its constructor. In this case, doing so would allow users to sidestep validation and `send` invalid emails, but you're not always building an API like this, so exporting constructors can be useful. With a type that has several constructors, you may also only want to export some of them.

Constructing Types

The `type alias` keyword combination gives a new name for a type, but the `type` keyword in isolation declares a new type. Let's examine one of the most fundamental of these types: `Maybe`

```
type Maybe a
    = Just a
    | Nothing
```

The first thing to note is that the `Maybe` type is declared with a **type variable** of `a`. The second thing to note is the pipe character, `|`, which signifies "or". In other words, something of type `Maybe a` is *either* `Just a` *or* `Nothing`.

When you write the above code, `Just` and `Nothing` come into scope as *value-constructors*, and `Maybe` comes into scope as a *type-constructor*. These are their signatures:

```
Just : a -> Maybe a

Nothing : Maybe a

Maybe : a -> Maybe a -- this can only be used in type signatures
```

Because of the *type variable* `a`, any type can be "wrapped inside" of the `Maybe` type. So, `Maybe Int`, `Maybe (List String)`, or `Maybe (Maybe (List Html))`, are all valid types. When destructuring any `type` value with a `case` expression, you must account for each possible instantiation of that type. In the case of a value of type `Maybe a`, you have to account for both the `Just a` case, and the `Nothing` case:

```
thing : Maybe Int
thing =
    Just 3

blah : Int
blah =
    case thing of
        Just n ->
            n

        Nothing ->
            42

-- blah = 3
```

Try writing the above code without the `Nothing` clause in the `case` expression: it won't compile. This is what makes the `Maybe` type-constructor a great pattern for expressing values that may not exist, as it forces you to handle the logic of when the value is `Nothing`.

The Never type

The `Never` type cannot be constructed (the `Basics` module hasn't exported its **value constructor** and hasn't given you any other function that returns `Never` either). There is no value `never : Never` or a function `createNever : ?? -> Never`.

This has its benefits: you can encode in a type system a possibility that can't happen. This can be seen in types like `Task Never Int` which guarantees it will succeed with an `Int`; or `Program Never` that will not take any parameters when initializing the Elm code from JavaScript.

Special Type Variables

Elm defines the following special type variables that have a particular meaning to the compiler:

- **`comparable`**: Comprised of `Int`, `Float`, `Char`, `String` and tuples thereof. This allows the use of the `<` and `>` operators.

  Example: You could define a function to find the smallest and largest elements in a list (`extent`). You think what type signature to write. On one hand, you could write `extentInt : List Int -> Maybe (Int, Int)` and `extentChar : List Char -> Maybe (Char, Char)` and another for `Float` and `String`. The implementation of these would be the same:

  ```
  extentInt list =
    let
      helper x (minimum, maximum) =
        ((min minimum x), (max maximum x))
    in
      case list of
        [] ->
          Nothing
        x :: xs ->
          Just <| List.foldr helper (x, x) xs
  ```

  You might be tempted to simply write `extent : List a -> Maybe (a, a)`, but the compiler will not let you do this, because the functions `min` and `max` are not defined for these types (NB: these are just simple wrappers around the `<` operator mentioned above). You can solve this by defining `extent : List comparable -> Maybe (comparable, comparable)`. This allows your solution to be *polymorphic*, which just means that it will work for more than one type.

- **`number`**: Comprised of `Int` and `Float`. Allows the use of arithmetic operators except division. You can then define for example `sum : List number -> number` and have it work for both ints and floats.

- **`appendable`**: Comprised of `String`, `List`. Allows the use of the `++` operator.

- **`compappend`**: This sometimes appears, but is an implementation detail of the compiler. Currently this can't be used in your own programs, but is sometimes mentioned.

Note that in a type annotation like this: `number -> number -> number` these all refer to the same type, so passing in `Int -> Float -> Int` would be a type error. You can solve this by adding a suffix to the type variable name: `number -> number' -> number''` would then compile fine.

There is no official name for these, they are sometimes called:

- Special Type Variables

- Typeclass-like Type Variables
- Pseudo-typeclasses

This is because they work like Haskell's Type Classes, but without the ability for the user to define these.

Read Types, Type Variables, and Type Constructors online:
https://riptutorial.com/elm/topic/2648/types--type-variables--and-type-constructors

Credits

S. No	Chapters	Contributors
1	Getting started with Elm Language	2426021684, alejosocorro, AnimiVulpis, Community, Douglas Correa, gabrielperales, gar, halfzebra, Jakub Hampl, jmite, JustGage, lonelyelk, Martin Janiczek, mrkovec, thSoft, Zimm i48
2	Backend Integration	lonelyelk
3	Collecting Data: Tuples, Records and Dictionaries	halfzebra, Martin Janiczek, Mr. Baudin
4	Custom JSON Decoders	Khaled Jouda
5	Debugging	AnimiVulpis, bdukes, Jonathan de M., Martin Janiczek, Nicholas Montaño
6	Functions and Partial Application	Art Yerkes, halfzebra, lonelyelk, Martin Janiczek, Nicholas Montaño, Ryan Plant, Will White
7	Json.Decode	ivanceras, Jonathan de M., lonelyelk, Matthew Rankin
8	Lists and Iteration	2426021684, AnimiVulpis, jmite, lonelyelk, Martin Janiczek, Nicholas Montaño, Zimm i48
9	Making complex update functions with ccapndave/elm-update-extra	Mateus Felipe
10	Pattern Matching	Gerald Kaszuba, Jakub Hampl, Tosh
11	Ports (JS interop)	Adam Bowen, gabrielperales, halfzebra, Martin Janiczek, Nicholas Montaño
12	Subscriptions	lonelyelk, mrkovec, Tosh
13	The Elm Architecture	AnimiVulpis, halfzebra, mrkovec, Ryan Plant, vlad_o, Zimm i48
14	Types, Type Variables, and Type Constructors	Art Yerkes, bright-star, halfzebra, Jakub Hampl, Joseph Weissman, Martin Janiczek, Nicholas Montaño, Zimm i48