

Elm

IN ACTION

Richard Feldman



MEAP



MANNING



**MEAP Edition
Manning Early Access Program
Elm in Action
Version 7**

Copyright 2018 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thanks for buying the MEAP of *Elm in Action*! I hope you find this a wonderfully practical journey into a delightful language.

As an early adopter of Elm, I felt like a wide-eyed sprinter barreling down an exciting path that had not yet been paved. Now that I've had time to catch my breath, I'm eager to help pave that road for future travelers. My goal for this book is for you to enjoy learning Elm even more than I did...and I loved it so much I'm writing a book about it!

More than anything, I hope this book gives you the confidence to use Elm professionally. As much as I enjoyed using Elm on a side project, it's brought me even more joy at work. When I think back on what motivated my team to try it, it was the practical benefits: reliability, a quicker development loop, and a lower maintenance burden.

The most authentic way I know to convey the benefits of building an application in Elm is...well, to build an application in Elm. That's exactly what we'll be doing over the course of this book: developing an application from start to finish, learning the language along the way. By the end you'll have built, refactored, and tested an Elm code base. If using it at work sounds appealing, you'll be able to convey its benefits based on firsthand experience!

Throughout the MEAP process I intend both to release new chapters as I finish them, and to revise past chapters based on feedback from readers like you. I'd love it if you could leave some comments in the Author Online Forum. All feedback is helpful! Mentioning which parts felt difficult tells me where to focus my revision time, and mentioning your favorite parts tells me what not to change.

Thanks again, and enjoy!

—Richard Feldman

brief contents

PART 1: GETTING STARTED

- 1 Welcome to Elm*
- 2 Your First Elm Application*
- 3 Compiler as Assistant*

PART 2: PRODUCTION-GRADE ELM

- 4 Talking to Servers*
- 5 Talking to JavaScript*
- 6 Testing*
- 7 Data Modeling*

PART 3: BUILDING BIGGER

- 8 Single-Page Applications*
- 9 Scaling Elm Code*
- 10 Performance Optimization*

Appendix A: Getting Set Up

1

Welcome to Elm

This chapter covers

- How and why to introduce Elm to a project
- Using `elm repl`
- Building expressions
- Writing and importing functions
- Working with collections

Back in 2014 I set out to rewrite a side project, and ended up with a new favorite programming language. Not only was the rewritten code faster, more reliable, and easier to maintain, writing it was the most fun I'd had in over a decade writing code. Ever since that project, I've been hooked on Elm.

The rewrite in question was a writing application I'd built even longer ago, in 2011. Having tried out several writing apps over the course of writing a novel, and being satisfied with none, I decided to scratch my own itch and build my dream writing app. I called it Dreamwriter.

For those keeping score: yes, I was indeed writing code in order to write prose better.

Things went well at first. I built the basic Web app, started using it, and iterated on the design. Months later I'd written over fifty thousand words in Dreamwriter. If I'd been satisfied with that early design, the story might have ended there. However, users always want a better experience...and when the user and the developer are the same person, further iteration is inevitable.

The more I revised Dreamwriter, the more difficult it became to maintain. I'd spend hours trying to reproduce bugs that knocked me out of my writing groove. At some point the copy and paste functions stopped working, and I found myself resorting to the browser's developer tools whenever I needed to move paragraphs around.

Right around when I'd decided to scrap my unreliable code base and do a full rewrite, a blog post crossed my radar. After reading it I knew three things:

1. The Elm programming language compiled to JavaScript, just like Babel or CoffeeScript. (I already had a compile step in my build script, so this was familiar territory.)
2. Elm used the same rendering approach as React.js—which I had recently grown to love—except Elm had rendering benchmarks that outperformed React's!
3. Elm's compiler would catch a lot of the errors I'd been seeing before they could harm me in production. I did not yet know just how many it would catch.

I'd never built anything with a functional programming language like Elm before, but I decided to take the plunge. I didn't really know what I was doing, but the compiler's error messages kept picking me up whenever I stumbled. Eventually I got the revised version up and running, and began to refactor.

The refactoring experience blew me away. I revised the Elm-powered Dreamwriter gleefully, even recklessly—and no matter how dramatic my changes, the compiler always had my back. It would point out whatever corner cases I'd missed, and I'd go through and fix them. As soon as the code compiled, lo and behold, everything worked again. I felt *invincible*.

I related my Elm experience to my coworkers at NoRedInk, and they were curious but understandably cautious. How could we find out if the team liked it without taking a big risk? A full rewrite may have been fine for Dreamwriter, but it would have been irresponsible to attempt that for our company's entire front-end.

So we introduced Elm gently, by rewriting just one portion of one production feature in Elm. It went well, so we did a bit more. And then more.

Today our front-end is as Elm-powered as we can make it, and our team has never been happier. Our test suites are smaller, yet our product is more reliable. Our feature set has grown more complex, yet refactoring remains delightful. We swap stories with other companies using Elm about how long our production code has run without throwing a runtime exception.

In this book we'll explore all of these benefits.

After learning some basics, we'll build an Elm Web application the way teams typically do: ship a basic version that works, but which has missing features and some technical debt. As we advance through the chapters, we'll expand and refactor this application, adding features and paying off technical debt as we learn more about Elm. We'll debug our business logic. We'll even interoperate with JavaScript, tapping into its vast library ecosystem without compromising our Elm experience. By the end of the book we will have transformed our application into a more featureful product, with a more maintainable code base, than the one we initially shipped.

With any luck, we'll have a lot of fun doing it.

Welcome to Elm!

1.1 How Elm Fits In

Elm can be used either as a replacement for in-browser JavaScript code, or as a complement to it. You write some `.elm` files, run them through Elm’s compiler, and end up with plain old `.js` files that the browser runs as normal. If you have separate stylesheets that you use alongside JavaScript, they’ll work the same way alongside Elm.

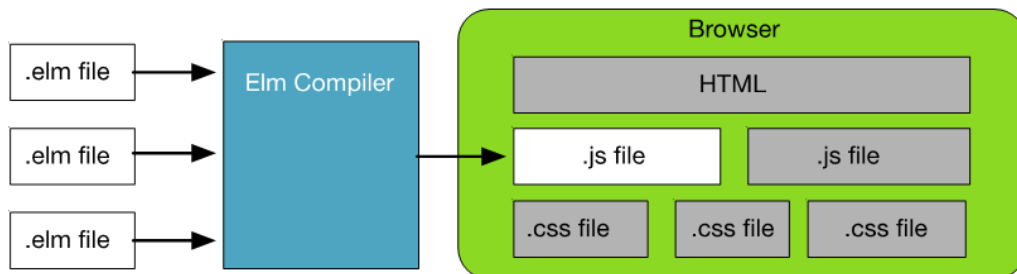


Figure 1.1 Elm files are compiled to plain old JavaScript files

The appropriate Elm-to-JavaScript ratio can vary by project. Some projects may want primarily JavaScript and only a touch of Elm for business logic or rendering. Others may want a great deal of Elm but just a pinch of JavaScript to leverage its larger ecosystem. No single answer applies to every project.

What distinguishes Elm from JavaScript is **maintainability**.

Handwritten JavaScript code is notoriously prone to runtime crashes like “undefined is not a function.” In contrast, Elm code has a reputation for never throwing runtime exceptions in practice. This is because Elm is built around a small set of simple primitives like expressions, immutable values, and managed effects. That design lets the compiler identify lurking problems just by scanning your source code. It reports these problems with such clarity that it has earned a reputation for user-friendliness even among programming legends.

That should be an inspiration for every error message.

—John Carmack, after seeing one of Elm’s compiler errors

Having this level of compiler assistance makes Elm code dramatically easier to refactor and debug, especially as code bases grow larger. There is an up-front cost to learning and adopting Elm, but you reap more and more maintainability benefits the longer the project remains in active development.

TIP Most teams that use Elm in production say they used a “planting the seed” approach. Instead of waiting for a big project where they could build everything in Elm from the ground up, they rewrote a small part of their existing JavaScript code base in Elm. This was low-risk and could be rolled back if things did not go as planned,

but having that small seed planted in production meant they could grow their Elm code at a comfortable pace from then on.

Although Elm is in many ways a simpler language than JavaScript, it's also much younger. This means Elm has fewer off-the-shelf solutions available for any given problem. Elm code can interoperate with JavaScript code to piggyback the larger JavaScript library ecosystem, but Elm's design differs enough from JavaScript's that incorporating JavaScript libraries takes effort.

Balancing these tradeoffs depends on the specifics of a given project. Let's say you're on a team where people are comfortable with JavaScript but are new to Elm. Here are some projects I'd expect would benefit from learning and using Elm:

- Feature-rich Web applications whose code bases are large or will grow large
- Individual features that will be revised and maintained over an extended period of time
- Projects where most functionality comes from in-house code, not off-the-shelf libraries

In contrast, I might choose a more familiar language and toolset for projects like these:

- Time-crunched projects where learning a language is unrealistic given the deadline
- Projects that will consist primarily of gluing together off-the-shelf components
- Quick proof-of-concept prototypes that will not be maintained long-term

We'll explore these tradeoffs in more detail throughout the course of the book.

1.2 Expressions

To get our feet wet with Elm, let's tap into one of the most universal traits across the animal kingdom: the innate desire to *play*.

Researchers have developed many theories as to why we play, including to learn, to practice, to experiment, and of course for the pure fun of it. These researchers could get some high-quality data by observing a member of the *homo sapiens programmerus* species in its natural environment for play—the Read-Eval-Print Loop, or REPL.

You'll be using Elm's REPL to play as you take your first steps as an Elm programmer.

1.2.1 Using elm repl

The Elm Platform includes a nice REPL called `elm repl`, so if you have not installed the Elm Platform yet, head over to Appendix A to get hooked up.

Once you're ready, enter `elm repl` at the terminal. You should see this prompt:

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

Alexander Graham Bell invented the telephone over a century ago. There was no customary greeting back then, so Bell suggested one: lift the receiver and bellow out a rousing “Ahoy!” Thomas Edison later proposed the alternative “Hello,” which stuck, and today programmers everywhere append “World” as the customary way to greet a new programming language.

Let’s spice things up a bit, shall we? Enter this at the prompt.

```
> "Ahoy, World!"
```

You should see this response from `elm repl`:

```
"Ahoy, World!" : String
```

Congratulations, you are now an Elm programmer!

NOTE To focus on the basics, for the rest of this chapter we’ll omit the type annotations that `elm repl` prints. For example, the previous code snippet would have omitted the `: String` portion of `"Ahoy, World!" : String`. We’ll get into these annotations in Chapter 3.

If you’re the curious sort, by all means feel free to play as we continue. Enter things that occur to you, and see what happens! Whenever you encounter an error you don’t understand yet, picture yourself as a tiger cub building intuition for physics through experimentation: adorable for now, but powerful in time.

1.2.2 Building Expressions

Let’s rebuild our “Ahoy, World!” greeting from two parts, and then play around from there. Try entering these into `elm repl`.

Listing 1.1 Combining Strings

```
> "Ahoy, World!"
"Ahoy, World!"

> "Ahoy, " ++ "World!"
"Ahoy, World!"

> "Pi is " ++ String.fromFloat pi ++ " (give or take)" #A
"Pi is 3.141592653589793 (give or take)"
```

#A `String.fromFloat` is a standalone function, not a method. We will cover it later.

In Elm, we use the `++` operator to combine strings, instead of the `+` operator JavaScript uses. At this point you may be wondering: Does Elm even have a `+` operator? What about the other arithmetic operators?

Let’s find out by experimenting in `elm repl`!

Listing 1.2 Arithmetic Expressions

```
> 1234 + 103
```

```

1337
> 12345 - (5191 * -15) #A
90210

> 2 ^ 11
2048

> 49 / 10
4.9

> 49 // 10 #B
4

> -5 % 2 #C
1

```

#A Nest expressions via parentheses

#B Integer division (decimals get truncated)

#C Remainder after integer division

Sure enough, Elm has both a ++ and a + operator. They are used for different things:

- The ++ operator is for appending. Using it on a number is an error.
- The + operator is for addition. It can *only* be used on numbers.

You will see this preference for **being explicit** often in Elm. If two operations are sufficiently different—in this case, adding and appending—Elm implements them separately, so each implementation can **do one thing well**.

STRINGS AND CHARACTERS

Elm also distinguishes between strings and the individual UTF-8 *characters* that comprise them. Double quotes in Elm represent string literals, just like in JavaScript, but single quotes in Elm represent character literals.

Table 1.1 Strings and Characters

Elm Literal	Result
"a"	a string with a length of 1
'a'	a single character
"abc"	a string with a length of 3
'abc'	error: character literals must contain exactly 1 character
""	an empty string
' '	error: character literals must contain exactly 1 character

COMMENTS

There are two ways to write comments in Elm:

- Use `--` for single-line comments (like `//` in JavaScript)
- Use `{-` to begin a multi-line comment, and `-}` to end it (like `/*` and `*/` in JavaScript)

Let's see these in action!

Listing 1.3 Characters, Comments, and Named Values

```
> 'a' -- This is a single-line comment. It will be ignored. #A
'a'

> "a" {- This comment could span multiple lines. -} #B
"a"

> milesPerHour = 88 #C
88

> milesPerHour
88
```

```
#A JavaScript comment: //
#B JavaScript comment: /* ... */
#C JavaScript: const milesPerHour = 88;
```

ASSIGNING NAMES TO VALUES

In the last two lines of code above, we did something new: we assigned the name `milesPerHour` to the value `88`.

NOTE Once we assign a name to a value, that name cannot be later reassigned to a different value in the same scope. Assignment in Elm works like JavaScript's `const` keyword, as opposed to `var` or `let`.

There are a few things to keep in mind when assigning names to values.

- The name must begin with a lowercase letter. After that it can be a mix of letters, numbers, and underscores.
- By convention, all letters should be in one uninterrupted sequence. For example, `map4` is a reasonable name, but `map4ever` is not, as the sequence of letters is interrupted by the 4.
- Because of the previous two rules, you should never use `snake_case` or `SCREAMING_SNAKE_CASE` to name values. Use `camelCase` instead.
- If you absolutely must know whether the compiler will accept `some_raD__THiNG__` as a valid name, remember: what happens in `elm repl` stays in `elm repl`.

ASSIGNING NAMES TO EXPRESSIONS

Not only can you assign names to literal values, you can also assign them to *expressions*.

DEFINITION An *expression* is anything that evaluates to a single value.

Here are some expressions we've seen so far.

Expression	Evaluates to
"Ahoy, " ++ "World!"	"Ahoy, World!"
2^{11}	2048
pi	3.141592653589793
42	42

NOTE Since an expression is anything that evaluates to a value, literal values like "Ahoy, World!" and 42 are expressions too—just expressions that have already been fully evaluated.

Expressions are the basic building block of Elm applications. This is different from JavaScript, which offers many features as *statements* instead of expressions.

Consider these two lines of JavaScript code.

```
label = (num > 0) ? "positive" : "negative" // ternary expression
label = if (num > 0) { "positive" } else { "negative" } // if-statement
```

The first line is ternary *expression*. Being an expression, it evaluates to a value, and JavaScript happily assigns that value to `label`.

The second line is an *if-statement*, and since statements do not evaluate to values, trying to assign it to `label` yields a syntax error.

This distinction does not exist in Elm, as Elm programs express logic using expressions only. As such, Elm has *if-expressions* instead of *if-statements*. As we will see in Chapter 2, every Elm application is essentially one big expression built up from many smaller ones!

1.2.3 Booleans and Conditionals

There aren't many booleans out there—just the two, really—and working with them in Elm is similar to working with them in JavaScript. There are a few differences, though.

- You write `True` and `False` instead of `true` and `false`
- You write `/=` instead of `!==`
- To negate values, you use Elm's `not` function instead of JavaScript's `!` prefix

Let's try them out!

Listing 1.4 Boolean Expressions

```
> pi == pi #A
True #B

> pi /= pi #C
False #D

> not (pi == pi) #E
False

> pi <= 0 || pi >= 10
False

> 3 < pi && pi < 4 #F
True
```

```
#A JavaScript: pi === pi
#B JavaScript: true
#C JavaScript: pi !== pi
#D JavaScript: false
#E JavaScript: !(pi === pi)
#F 3 < pi < 4 would be an error
```

Now let's say it's a lovely afternoon at the North Pole, and we're in Santa's workshop writing a bit of UI logic to display how many elves are currently on vacation. The quick-and-dirty approach would be to add the string " elves" after the number of vacationing elves, but then when the count is 1 we'd display "1 elves", and we're better than that.

Let's polish our user experience with the *if-expression* shown in Figure 1.2.

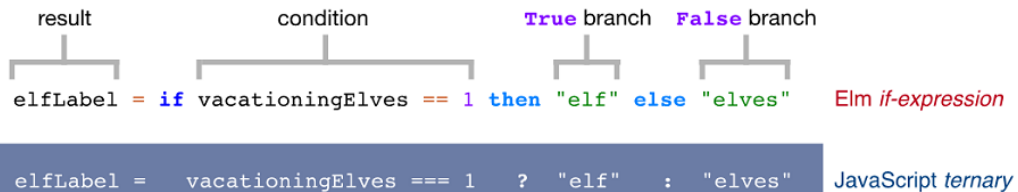


Figure 1.2 Comparing an Elm *if-expression* to a JavaScript *ternary*

Like JavaScript ternaries, Elm *if-expressions* require three ingredients:

1. A condition
1. A branch to evaluate if the condition passes
2. A branch to evaluate otherwise

Each of these ingredients must be expressions, and the whole *if-expression* evaluates to the result of whichever branch got evaluated. You'll get an error if any of these three ingredients are missing, so make sure to specify an `else` branch every time!

NOTE There is no such thing as “truthiness” in Elm. Conditions can be either `True` or `False`, and that’s it. Life is simpler this way.

Now let’s say we modified our pluralization conditional to include a third case:

- If we have one Elf, evaluate to `"elf"`
- Otherwise, if we have a positive number of elves, evaluate to `"elves"`
- Otherwise, we must have a negative number of elves, so evaluate to `"anti-elves"`

In JavaScript you may have used `else if` to continue branching conditionals like this. It’s common to use `else if` for the same purpose in Elm, but it’s worth noting that `else if` in Elm is nothing more than a stylish way to combine the concepts we learned a moment ago.

Check it out!

Listing 1.6 Using `else if`

```
if elfCount == 1 then                #A
  "elf"                              #A
else                                 #A
  (if elfCount >= 0 then "elves" else "anti-elves") #A

if elfCount == 1 then                #B
  "elf"                              #B
else (if elfCount >= 0 then          #B
  "elves"                            #B
else                                 #B
  "anti-elves")                     #B

if elfCount == 1 then                #C
  "elf"                              #C
else if elfCount >= 0 then           #C
  "elves"                            #C
else                                 #C
  "anti-elves"                       #C
```

#A Use an *if-expression* inside `else`

#B Rearrange some whitespace

#C Drop the parentheses

This works because the `else` branch of an *if-expression* must be an expression, and it just so happens that *if-expressions* themselves are expressions. As shown in Figure 1.3, all it takes is putting an *if-expression* after another one’s `else`, and *voilà!* Additional branching achieved.

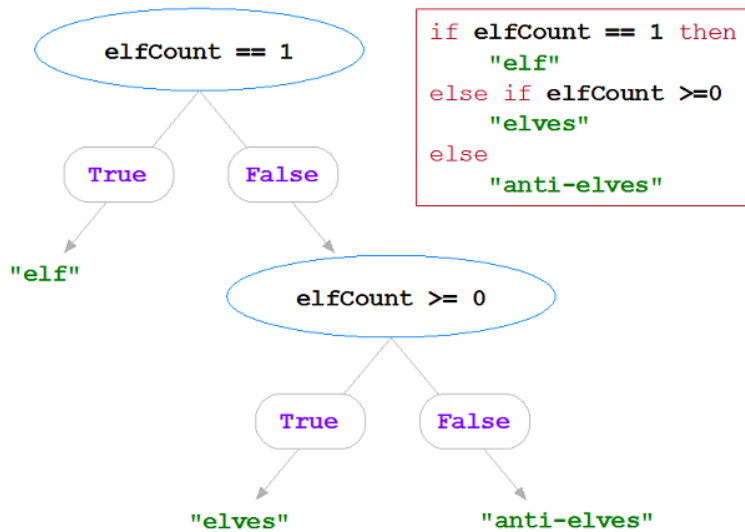


Figure 1.3 The *else if* technique: use an *if-expression* as the *else* branch of another *if-expression*

Nesting expressions is a recurring theme in Elm, and we'll see plenty more recipes like `else if` throughout the book.

Chapter 3 will add a powerful new conditional to our expression toolbox, one which has no analogue in JavaScript: the *case-expression*.

1.3 Functions

Earlier we wrote this expression:

```
elfLabel = if vacationingElves == 1 then "elf" else "elves"
```

Suppose it turns out a general-purpose singular/plural labeler would be really useful, and we want to reuse similar logic across the code base at Santa's Workshop. Search results might want to display "1 result" and "2 results" as appropriate, for example.

We can write a *function* to make this pluralization logic easily reusable.

DEFINITION Elm *functions* represent reusable logic. They are not objects. They have no fields, no prototypes, and no ability to store state. All they do is accept values as arguments, and then return a value.

If you thought expressions would be a recurring theme in Elm, wait 'til you see functions!

1.3.1 Defining Functions

Let's define our first function: `isPositive`. It will take a number and then:

- Return `True` if the number is greater than zero
- Return `False` otherwise

We can define `isPositive` in `elm repl` and try it out right away.

Listing 1.7 Defining a function

```
> isPositive num = num > 0    #A
<function>

> isPositive 2                #B
True

> (isPositive 2)              #C
True

> isPositive (2 - 10)         #D
False
```

#A JavaScript: `function isPositive(num) { return num > 0; }`

#B JavaScript: `isPositive(2)`

#C JavaScript: `(isPositive(2))`

#D JavaScript: `isPositive(2 - 10)`

As you can see, in Elm we put the function parameter name before the `=` sign. We also don't surround the function body with `{ }`. And did you notice the `return` keyword is nowhere to be seen? That's because Elm doesn't have one! In Elm, a function body is a single expression, and since an expression evaluates to a single value, Elm uses that value as the function's return value. This means all Elm functions return values!

For our `isPositive` function, the expression `num > 0` serves as the function's body, and provides its return value.

Refactoring out an early return

In JavaScript, `return` is often used to exit a function early. This is harmless when used responsibly, but can lead to unpleasant surprises when used in the middle of large functions. Elm does not support these unpleasant surprises, because it has no `return` keyword.

Let's refactor the early `return` out of this function:

```
function capitalize(str) {
  if (!str) {
    return str;    #A
  }

  return str[0].toUpperCase() + str.slice(1);
}
```

#A Early return

Without making any other changes, we can refactor this early `return` into a ternary:

```
function capitalize(str) {
  return !str ? str : str[0].toUpperCase() + str.slice(1);
}
```

Poof! There it goes. Since JavaScript's ternaries are structurally similar to Elm's *if-expressions*, this code is now much more straightforward to rewrite in Elm. More convoluted JavaScript functions may require more steps than this, but it is always possible to untangle them into plain old conditionals.

Removing an early `return` is one of many quick refactors you can do to ease the transition from legacy JavaScript to Elm, and we'll look at more of them throughout the book. When doing these, do not worry if the intermediate JavaScript code looks ugly! It's intended to be a stepping stone to nicer Elm code, not something to be maintained long-term.

Let's use what we just learned to generalize our previous `elf-labeling` expression into a reusable `pluralize` function. Our function this time will have a longer definition than last time, so let's use multiple lines to give it some breathing room. In `elm repl`, you can enter multiple lines by adding `\` to the end of the first line and indenting the next line.

NOTE Indent with spaces only! Tab characters are syntax errors in Elm.

Listing 1.8 Using multiple REPL lines

```
> pluralize singular plural count = \
|   if count == 1 then singular else plural   #A
<function>

> pluralize "elf" "elves" 3   #B
"elves"

> pluralize "elf" "elves" (round 0.9)   #C
"elf"
```

#A Don't forget to indent!

#B No commas between arguments!

#C (round 0.9) returns 1

When passing multiple arguments to an Elm function, separate the arguments with whitespace and not commas. That last line of code is an example of passing the result of one function call, namely `round 0.9`, as an argument to another function. (Think about what would happen if we did not put parentheses around `(round 0.9)`...how many arguments would we then be passing to `pluralize`?)

1.3.2 Importing Functions

So far we've only used basic operators and functions we wrote ourselves. Now let's expand our repertoire of functions by using one from an external module.

DEFINITION A *module* is a named collection of Elm functions and other values.

The `String` module is one of the core modules that ships with Elm. Additional modules can be obtained from Elm's official package repository, copy-pasting code from elsewhere, or through

a back-alley rendezvous with a shadowy figure known as Dr. Deciduous. Chapter 4 will cover how to do the former, but neither the author nor Manning Publications endorses obtaining Elm modules through a shadowy back-alley rendezvous.

Let's import the `String` module and try out some of its functions.

Listing 1.9 Importing functions

```
> String.toLowerCase "Why don't you make TEN louder?"
"why don't you make ten louder?"

> String.toUpperCase "These go to eleven."
"THESE GO TO ELEVEN."

> String.fromFloat 44.1
"44.1"

> String.fromInt 1337
"1337"
```

The `String.fromFloat` and `String.fromInt` functions convert Floats (numbers that may be fractions) and Integers (numbers that may not be fractions) to strings.

In JavaScript, both are handled by the catch-all `toString` method, whereas Elm generally uses separate functions to convert between different types of values. This way, Elm can (and will) give an error if you accidentally call `String.fromFloat` on a function instead of a number, rather than cheerfully displaying gibberish to a very confused user who was expecting to see their account balance.

USING FUNCTIONS FROM THE STRING MODULE

Observant readers may note a striking resemblance between Elm's `String.toUpperCase` function and the `toUpperCase()` method one finds on JavaScript strings. This is the first example of a pattern we will encounter many times!

JavaScript has several ways of organizing string-related functionality: fields on a string, methods on a string, or methods on the `String` global itself.

In contrast, Elm strings have neither fields nor methods. The `String` module houses the standard set of string-related features, and exposes them in the form of plain old functions like `toLowerCase` and `toUpperCase`.

Table 1.2 String Functionality Comparison

JavaScript	Elm
"storm".length	<code>String.length</code> "storm"
"dredge".toUpperCase()	<code>String.toUpperCase</code> "dredge"
<code>String.fromCharCode(something)</code>	<code>String.fromCharCode</code> something

Not only is this organizational pattern consistent within the `String` module, it's consistent across Elm. Want a function related to Sets? Look no further than the functions in the `Set` module. Debugging functions? Hit up the `Debug` module.

Methods are never the answer in Elm; over here it's all vanilla functions, all the time.

TIP Complete documentation for `String`, `Set`, `Debug`, and other tasty modules can be found in the *core* section of the package.elm-lang.org website.

We'll learn more about modules in the coming chapters, including how to write our own!

USING `String.filter` TO FILTER OUT CHARACTERS

Another useful function in the `String` module is `filter`. It lets us filter out unwanted characters from a string, such as non-numeric digits from a phone number.

To do this, we'll pass `filter` a function which specifies which characters to keep. The function will take a single character as an argument and return `True` if we should keep that character or `False` if we should chuck it. Figure 1.4 illustrates using `String.filter` to remove dashes from a US telephone number.

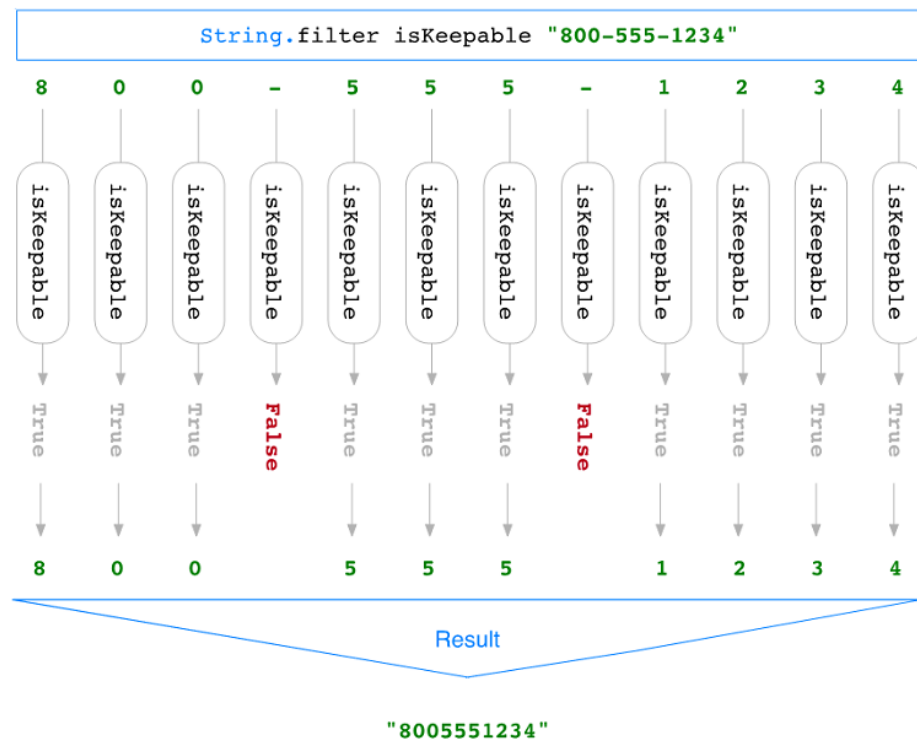


Figure 1.4 Using `String.filter` to remove dashes from a US phone number

Elm functions are first-class values that can be passed around just like any other value. This lets us provide `filter` with the function it expects by defining that function and then passing it in as a plain old argument.

Listing 1.10 Filtering with a named function

```
> isKeepable character = character /= '-' #A
<function>

> isKeepable 'z'
True

> isKeepable '-'
False

> String.filter isKeepable "800-555-1234" #B
"8005551234"
```

#A A function describing which characters to keep

#B Passing our function to `String.filter`

This normalizes telephone numbers splendidly. Alexander Graham Bell would be proud!

`String.filter` is one of the *higher-order functions* (that is, functions which accept other functions as arguments) that Elm uses to implement customizable logic like this.

1.3.3 Creating scope with *let-expressions*

Let's say we find ourselves removing dashes from phone numbers so often, we want to make a reusable function for it. We can do that with our trusty `isKeepable` function:

```
withoutDashes str = String.filter isKeepable str
```

This works, but in a larger Elm program, it might be annoying having `isKeepable` in the global scope like this. After all, its implementation is only useful to `withoutDashes`. Can we avoid globally reserving such a nicely self-documenting name?

Absolutely! We can scope `isKeepable` to the implementation of `withoutDashes` using a *let-expression*.

DEFINITION A *let-expression* adds locally scoped named values to an expression.

Figure 1.5 shows how we can implement `withoutDashes` using a single *let-expression*.

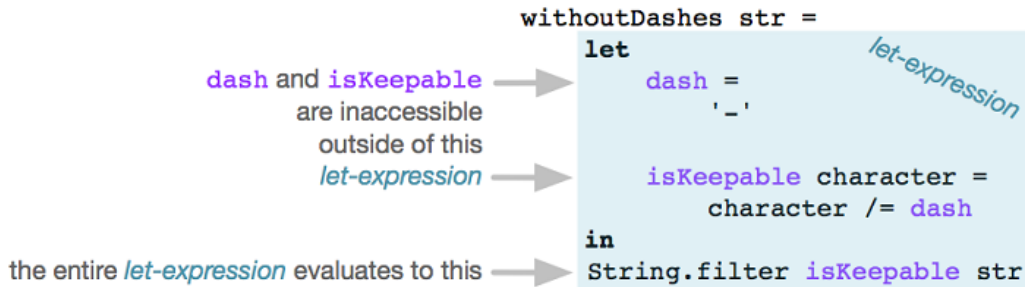


Figure 1.5 Anatomy of the wild *let-expression*

The above code does very nearly the same thing as entering the following in `elm repl`:

```
> dash = '-'
> isKeepable character = character /= dash
> withoutDashes str = String.filter isKeepable str
```

In both versions, the implementation of `withoutDashes` boils down to `String.filter isKeepable str`. The only difference between the two is the scope of `dash` and `isKeepable`.

- In the `elm repl` version above, `dash` and `isKeepable` are in the global scope.
- In Figure 1.5, `dash` and `isKeepable` are scoped locally to the *let-expression*.

You can mentally replace any *let-expression* with the part after its `in` keyword—in this case, `String.filter isKeepable str`. All the named values between `let` and `in` are intermediate values that are no longer in scope once the expression after `in` gets evaluated.

NOTE The indentation in Figure 1.5 is no accident! In a multiline *let-expression*, the `let` and `in` keywords must be at the same indentation level, and all other lines in the *let-expression* must be indented further than they are.

Anywhere you'd write a normal expression, you can swap in a *let-expression* instead. Because of this, you don't need to learn anything new to define locally-scoped named values inside function bodies, branches of *if-expressions*, or anywhere else.

Wherever you want some local scope, reach for a refreshing *let-expression*!

1.3.4 Anonymous Functions

Anonymous functions work like named functions, except they don't have a name.

Listing 1.11 compares named and anonymous functions in JavaScript and in Elm.

Listing 1.11 Named and Anonymous Functions

```
function area(w, h) { return w * h; } #A
function(w, h) { return w * h; } #B
```

```
area w h = w * h #C
```

```
\w h -> w * h #D
```

```
#A JavaScript named function
```

```
#B JavaScript anonymous function
```

```
#C Elm named function
```

```
#D Elm anonymous function
```

Elm's anonymous functions differ from named functions in three ways.

1. They have no names
2. They begin with a \
3. Their parameters are followed by -> instead of =

Once defined, anonymous functions and named functions work the same way; you can always use the one in place of the other. For example, the following do exactly the same thing:

```
isKeepable char = char /= '-'
isKeepable = \char -> char /= '-'
```

Let's use an anonymous function to call `String.filter` in one line instead of two, then see if we can improve the business logic! For example, we can try using `Char.isDigit` to cast a wider net, filtering out any non-digit characters instead of just dashes.

Listing 1.12 Filtering with anonymous functions

```
> String.filter (\char -> char /= '-') "800-555-1234"
"8005551234"

> String.filter (\char -> char /= '-') "(800) 555-1234"
"(800) 5551234" #A

> import Char
> String.filter (\char -> Char.isDigit char) "(800) 555-1234"
"8005551234" #B

> String.filter Char.isDigit "(800) 555-1234" #C
"8005551234"
```

```
#A Our simple filter fell short here
```

```
#B Much better!
```

```
#C Refactor of previous approach
```

Anonymous functions are often used with higher-order functions like `String.filter`.

1.3.5 Operators

So far we've seen functions such as `String.filter`, as well as operators such as `++`, `-`, and `==`. How do operators and functions relate?

As it turns out, Elm’s operators *are* functions! There are a few things that distinguish operators from normal functions:

- Operators must always accept exactly two arguments—no more, no fewer.
- Normal functions have names that begin with a letter. You typically call them by writing the name of the function followed by its arguments. This is *prefix-style* calling.
- Operators have names that contain neither letters nor numbers. You typically call them by writing the first argument, followed by the operator, followed by the second argument. This is *infix-style* calling.

Wrapping an operator in parentheses treats it as a normal function—*prefix-style* calling and all!

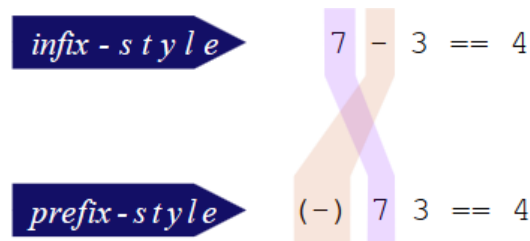


Figure 1.6 Calling the `(-)` operator in both *infix-style* and *prefix-style*

Let’s play with some operators in `elm repl`:

Listing 1.13 Operators are functions

```
> (/)
<function>

> divideBy = (/)
<function>

> 7 / 2    #A
3.5

> (/) 7 2  #B
3.5

> divideBy 7 2
3.5
```

#A *infix-style* calling
#B *prefix-style* calling

OPERATOR PRECEDENCE

Try entering an expression involving both arithmetic operators and `(==)` into `elm repl`:

```
> 3 + 4 == 8 - 1
True : Bool
```

Now consider how we'd rewrite this expression in *prefix-style*:

```
> (==) ((+) 3 4) ((-) 8 1)
True : Bool
```

Notice anything about the order in which these operators appear?

- Reading the *infix-style* expression from left to right, we see `+`, then `==`, and finally `-`.
- The *prefix-style* expression has a different order: first we see `==`, then `+`, and finally `-`.

How come? They get reordered because `(==)`, `(+)`, and `(-)` have different *precedence* values.

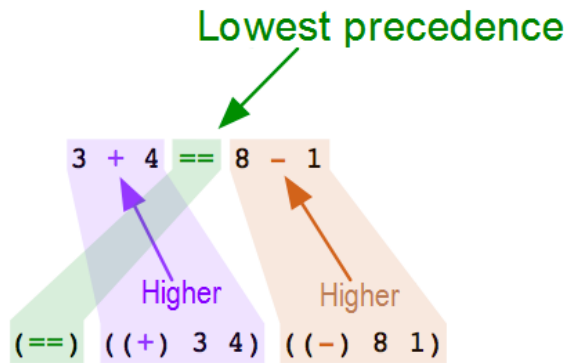


Figure 1.7 `(==)` gets evaluated after `(+)` and `(-)` because it has lower precedence

DEFINITION In any expression containing multiple operators, the operators with higher *precedence* get evaluated before those with lower precedence. This only applies to *infix-style* calls, as all *prefix-style* calls implicitly have the same precedence.

There isn't much formal documentation on operators' relative precedence values, but operators that appear in many programming languages (such as the `(==)`, `(+)`, and `(-)` operators) tend to work similarly in Elm to how they do everywhere else.

NORMAL FUNCTION CALLS HAVE TOP PRECEDENCE

Here are two ways to write the same thing:

```
> negate 1 + negate 5
-6

> (negate 1) + (negate 5)
-6
```

These two are equivalent because plain function calls have higher precedence than any operator. This means any time you want to pass the results of two plain function calls to an operator, you won't need to add any parentheses! You'll still get the result you wanted.

OPERATOR ASSOCIATIVITY

Besides precedence, the other factor that determines evaluation order for operators called in *infix-style* is whether the operators are *left-associative*, *right-associative*, or *non-associative*. Every operator is one of these.

An easy way to think about operator associativity is in terms of where the implied parentheses go. Infix expressions involving left-associative operators, such as arithmetic operators, have implied parentheses that cluster on the left:

Table 1.3 Implied parentheses for the `(-)` operator

Parentheses Shown	Expression	Result
none	<code>10 - 6 - 3</code>	1
assuming <i>left-associative</i>	<code>((10 - 6) - 3)</code>	1
assuming <i>right-associative</i>	<code>(10 - (6 - 3))</code>	7

If `(-)` were right-associative, `10 - 6 - 3` would have parentheses clustering on the right, meaning it would evaluate to `(10 - (6 - 3))` and the undesirable result of `10 - 6 - 3 == 7`. Good thing arithmetic operators are left-associative!

Non-associative operators cannot be chained together. For example, `foo == bar == baz` does not result in clustered parentheses, it results in an error!

1.4 Collections

Elm's most basic collections are lists, records, and tuples. Each has varying degrees of similarity to JavaScript's arrays and objects, but one way in which they differ from JavaScript collections is in that Elm collections are always *immutable*.

DEFINITION An *immutable* value cannot be modified in any way once created.

This is in contrast to JavaScript, where some values (like strings and numbers) are immutable, but collections (like arrays and objects) can be mutated.

1.4.1 Lists

An Elm list has many similarities to a JavaScript array.

- You can create one with a square bracket literal, e.g. `["one fish", "two fish"]`
- You can ask for its first element

- You can ask for its length
- You can iterate over its elements in various ways

An Elm list does have some differences, though.

- It is immutable
- It has no fields or methods. You work with it using functions from the `List` module.
- Because it is a *linked list*, you can ask for its first element, but not for other individual elements. (If you need to ask for elements at various different positions, you can first convert from an Elm `List` to an Elm `Array`. We'll discuss Elm Arrays in Chapter 3.)
- All elements in an Elm list must have a consistent type. For example, it can be a "list of numbers" or a "list of strings," but not a "list where strings and numbers intermingle." (Making a list containing both strings and numbers involves first creating wrapper elements for them, using a feature called *custom types* that we'll cover in Chapter 3.)

Although Elm supports both (immutable) lists and (also immutable) arrays, Elm lists are much more commonly used in practice.

Here are some examples of how Elm lists and JavaScript arrays differ.

Table 1.4 Contrasting JavaScript Arrays and Elm Lists

JavaScript Array	Elm List
<code>[1, 2, 3].length</code>	<code>List.length [1, 2, 3]</code>
<code>["one fish", "two fish"][0]</code>	<code>List.head ["one fish", "two fish"]</code>
<code>["one fish", "two fish"][1]</code>	No arbitrary position-based element access
<code>[1, 2].concat([3, 4])</code>	<code>[1, 2] ++ [3, 4]</code>
<code>[1, 2].push(3)</code>	Cannot be modified; use e.g. <code>append</code> instead
<code>[1, "Paper", 3]</code>	All elements in a list must have a consistent type

Let's focus on that last one. Why must all elements in an Elm list have a consistent type?

To understand how this requirement benefits us, let's delve into the `List.filter` function, which works like the `String.filter` function we used earlier.

We saw earlier that `String.filter` takes a function which returns `True` when the given character should be kept, and `False` when it should be dropped. `List.filter` differs only in that the function you provide doesn't necessarily receive characters—instead it receives elements from the list, whatever they may be.

Let's see that in action. Quick! To `elm repl`!

Listing 1.14 Filtering lists

```
> List.filter (\char -> char /= '-') [ 'Z', '-', 'Z' ] #A
['Z','Z']
```

```

> List.filter (\str -> str /= "-") [ "ZZ", "-", "Top" ] #B
["ZZ","Top"]

> import Char
> List.filter Char.isDigit [ '7', '-', '9' ] #C
['7','9']

> List.filter (\num -> num > 0) [ -2, -1, 0, 1, 2 ] #D
[1,2]

```

#A Same function we passed to `String.filter` earlier
#B Strings instead of characters
#C Works just like with `String.filter`
#D Keep only the positive numbers

Here's how we would rewrite that last line of code in JavaScript:

```

[ -2, -1, 0, 1, 2 ].filter(function(num) { return num > 0; })

```

This looks straightforward enough, but JavaScript arrays permit inconsistent element types. Without looking it up, can you guess what happens if we change it to the following?

```

[ -2, "0", "one", 1, "+02", "(3)" ].filter(function(num) { return num > 0; })

```

Will it crash? Will it happily return numbers? What about strings? It's a bit of a head-scratcher.

Because Elm requires consistent element types, this is a no-brainer: in Elm it would be an error. Even better, it would be an error at build time—meaning you can rest easy knowing whatever surprises would result from executing this code will not inflict pain on your users. Requiring consistent element types means all lists in Elm guarantee this level of predictability.

By the way, the above `filter()` call returns `[1, "+02"]`. (Like, *duh*, right?)

1.4.2 Records

We've now seen how JavaScript's mutable arrays resemble Elm's immutable lists. In a similar vein, JavaScript's mutable objects resemble Elm's immutable *records*.

DEFINITION A *record* is a collection of named fields, each with an associated value.

Whereas array and list literals between the two languages are syntactically identical, where JavaScript object literals use `:` to separate fields and values, Elm record literals use `=` instead.

Let's get a taste for some of their other differences.

JavaScript Object	Elm Record
<code>{ name: "Li", cats: 2 }</code>	<code>{ name = "Li", cats = 2 }</code>
<code>(({ name: "Li", cats: 2 })).cats</code>	<code>(({ name = "Li", cats = 2 })).cats</code>
<code>(({ name: "Li", cats: 2 }))["cats"]</code>	Fields can only be accessed directly, using a dot

<code>({ name: "Li", cats: 2 }).cats = 3</code>	Cannot be modified. (New cat? New record!)
<code>{ NAME: "Li", CATS: 2 }</code>	Field names can't start with uppercase letters
<code>({ name: "Li", cats: 2 }).__proto__</code>	No prepackaged fields, only the ones you define
<code>Object.keys({ name: "Li", cats: 5 })</code>	No listing of field names is available on demand
<code>Object.prototype</code>	Records have no concept of inheritance

Wow—compared to objects, records sure don't do much! It's like all they do is sit around holding onto the data we gave them. (Yep.) Personally I've found Elm's records a welcome reprieve from the intricacies of JavaScript's `this` keyword.

RECORD UPDATES

Record updates let us concisely obtain a new record by copying the old one and changing only the specified values. (Since records are immutable, Elm will reuse values from the existing record to save time and memory, rather than copying *everything*.)

Let's use this technique to represent someone obtaining an extra cat, going from `{ name = "Li", cats = 2 }` to `{ name = "Li", cats = 3 }` by way of a record update.

Listing 1.15 Record updates

```
> catLover = { name = "Li", cats = 2 }
{ name = "Li", cats = 2 }

> catLover
{ name = "Li", cats = 2 }

> withThirdCat = { catLover | cats = 3 } #A
{ name = "Li", cats = 3 }

> withThirdCat
{ name = "Li", cats = 3 }

> catLover                #B
{ name = "Li", cats = 2 }  #B

> { catLover | cats = 88, name = "LORD OF CATS" } #C
{ name = "LORD OF CATS", cats = 88 }
```

#A Record update syntax

#B Original record unmodified!

#C Update multiple fields (order doesn't matter)

Record updates let us represent this incremental evolution without mutating our records or recreating them from scratch. In Chapter 2 we'll represent our application state with a record, and use record updates to make changes based on user interaction.

1.4.3 Tuples

Lists let us represent collections of **varying size**, whose elements share a **consistent type**. Records let us represent collections of **fixed fields**, but whose corresponding values may have **varied types**.

Tuples introduce no new capabilities to this mix, as there is nothing a tuple can do that a record couldn't. Compared to records, though, what tuples bring to the party is conciseness.

DEFINITION A *tuple* is a record-like value whose fields are accessed by position rather than by name.

In other words, tuples are for when you want a record, but don't want to bother naming its fields. They are often used for things like key-value pairs where writing out `{ key = "foo", value = "bar" }` would add verbosity but not much clarity.

Let's try some out!

Listing 1.17 Using Tuples

```
> ( "Tech", 9 )
("Tech",9)

> Tuple.first ( "Tech", 9 ) #A
"Tech"

> Tuple.second ( "Tech", 9 ) #B
9
```

#A Return first element (only works on 2-element tuples)

#B Return second element (only works on 2-element tuples)

You can only use the `Tuple.first` and `Tuple.second` functions on tuples that contain two elements. If they have three elements, you can use *tuple destructuring* to extract their values.

DEFINITION *Tuple destructuring* extracts the values inside a tuple and assigns them to names in the current scope.

Let's use tuple destructuring to implement a function that takes a tuple of three elements.

Listing 1.18 Tuple Destructuring

```
> multiply3d ( x, y, z ) = x * y * z #A
<function>

> multiply3d ( 6, 7, 2 )
84

> multiply2d someTuple = let ( x, y ) = someTuple in x * y #B
<function>
```

#A Destructuring a tuple into three named values: x, y, and z

#B Destructuring a tuple inside a let-expression

As demonstrated in Listing 1.18, once you have named the values inside the tuple, you can use them just like you would any other named value.

TIP Mind the difference between a tuple and a parenthetical function call! `(foo, bar)` is a tuple, whereas `(foo bar)` is a call to the `foo` function passing `bar` as an argument. A simple mnemonic to remember the difference is “comma means tuple.”

Table 1.5 Comparing Lists, Records, and Tuples

List	Record	Tuple
Variable Length	Fixed Length	Fixed Length
Can Iterate Over	Cannot Iterate Over	Cannot Iterate Over
No Names	Named Fields	No Names
Immutable	Immutable	Immutable

Since any tuple can be represented using a record instead, Elm does not support tuples of more than three elements. For those situations, it’s better to use a record! When you only need two or three elements, though, choose tuples or records based on whichever would yield more readable code; their performance characteristics are equivalent.

1.5 Summary

We’re off to a fantastic start! First we discussed some of the toughest problems Web programmers face: crashing is too easy in JavaScript, and maintenance is too error-prone. Then we learned how Elm addresses these problems, with a design that prioritizes maintainability and a helpful compiler that catches would-be runtime exceptions before they can cause user pain. From there we dove in and wrote our first Elm code in `elm repl`.

Here is a brief review of things we covered along the way.

- The `++` operator combines strings, whereas the `+` operator is for addition only.
- Double quotes refer to strings. Single quotes refer to individual UTF-8 characters.
- *let-expressions* introduce scoped named values to an expression.
- There is no concept of “truthiness” in Elm, just `True` and `False`.
- `if foo /= bar then "different" else "same"` is an *if-expression*. Like JavaScript ternaries, *if-expressions* require an `else` branch and always evaluate to a value.
- Lists like `[3, 1, 4]` are immutable. Their elements must share a consistent type.
- `List.filter (\num -> num > 0) numbersList` returns a list containing all the positive numbers in the original `numbersList`.
- `catLover = { name = "Li", cats = 2 }` assigns a record to the name `catLover`.

Once assigned, names cannot be reassigned.

- `{ catLover | cats = 3 }` returns a new record that is the same as the `catLover` record, except the `cats` value is now 3.
- `(foo, bar)` deconstructs a tuple such as `(2, 3)`. In this example, `foo` would be 2 and `bar` would be 3.

Table 1.6 summarizes some of the differences between JavaScript and Elm.

Table 1.6 Differences between JavaScript and Elm

JavaScript	Elm
<code>// This is an inline comment</code>	<code>-- This is an inline comment</code>
<code>/* This is a block comment */</code>	<code>{- This is a block comment -}</code>
<code>true && false</code>	<code>True && False</code>
<code>"Ahoy, " + "World!"</code>	<code>"Ahoy, " ++ "World!"</code>
<code>"A spade" === "A spade"</code>	<code>"A spade" == "A spade"</code>
<code>"Calvin" !== "Hobbes"</code>	<code>"Calvin" /= "Hobbes"</code>
<code>Math.pow(2, 11)</code>	<code>2 ^ 11</code>
<code>Math.trunc(-49 / 10)</code>	<code>-49 // 10</code>
<code>n > 0 ? "positive" : "not"</code>	<code>if n > 0 then "positive" else "not"</code>
<code>nums.filter(function(n) { ... })</code>	<code>List.filter (\n -> n > 0) nums</code>
<code>function pluralize(s, p, c) { ... }</code>	<code>pluralize singular plural count = ...</code>

We also learned about several differences between plain functions and operators:

Table 1.7 Differences between plain functions and operators

Function	How to identify one	Calling style	Examples
Plain	Name begins with a letter	<i>prefix-style</i>	<code>negate</code> , <code>not</code> , <code>pluralize</code>
Operator	Name has no letters or numbers	<i>infix-style</i>	<code>(++)</code> , <code>(*)</code> , <code>(=)</code>

In Chapter 2 we'll expand on what we've learned here to create a working Elm application.

Let's go build something!

2

Your First Elm Application

This chapter covers

- Declaratively rendering a page
- Managing state with Model-View-Update
- Handling user interaction

Elm applications are built to last. They have a reputation for being scalable, easy to refactor, and difficult to crash unexpectedly. Since JavaScript applications have...well...a different reputation, it stands to reason that Elm must be doing things differently. And so it is!

Where each line of code in a JavaScript application can potentially result in a change or effect—like “*update that text!*” or “*send this to the server!*”—the code in an Elm application builds up a **description** of what the program should do in response to various inputs. Elm’s compiler translates this description into the appropriate JavaScript commands for the browser to run at the appropriate times, and the end user may have no idea Elm was involved at all.

In this chapter we’ll build our first Elm application: *Photo Groove*, a simple photo browsing Web app where users select thumbnails to view larger versions. We’ll create a user interface using declarative rendering and manage state using the Elm Architecture. By the end, we will have a fully functioning application—and a code base we can build on for the rest of the book!

2.1 Rendering a Page

Since the very early days of the Web, browsers have been translating HTML markup into a Document Object Model (or DOM for short) which represents the structure of the current page. The DOM consists of *DOM nodes*, and it’s only by changing these nodes that Web applications can modify the current page on the fly.

In this chapter we'll work with the two most common types of DOM nodes: *elements* and *text nodes*.

- *Elements* have a `tagName` (such as `"div"` or `"img"`), and may have child DOM nodes.
- *Text nodes* have a `textContent` property instead of a `tagName`, and are childless.

As Figure 2.1 shows, elements and text nodes can freely intermingle inside the DOM.

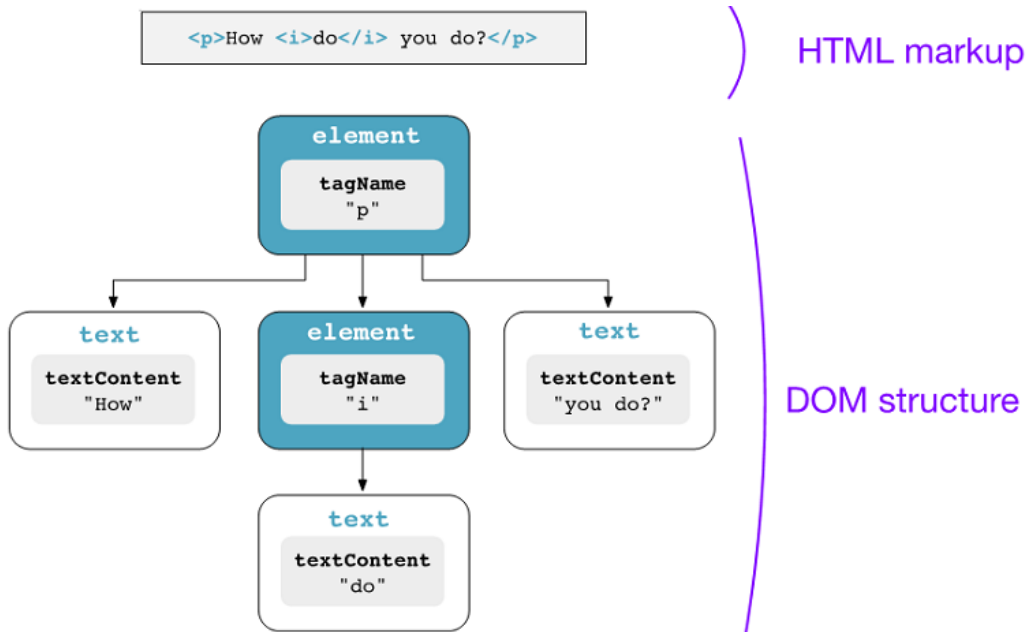


Figure 2.1 Intermingling element DOM nodes and text nodes

Here we've pulled back the curtain on the markup `<p>How <i>do</i> you do?</p>` to see that despite its two element tags—namely `<p>` and `<i>`—we are actually working with five DOM nodes here! The other three are not elements, but rather text nodes.

2.1.1 Describing a page using the `Html` Module

When describing how a page looks in Elm, we don't write markup. Instead, we call functions to create representations of DOM nodes. The most flexible of these functions is called `node`, and as Figure 2.2 shows, its arguments line up neatly with the analogous markup.

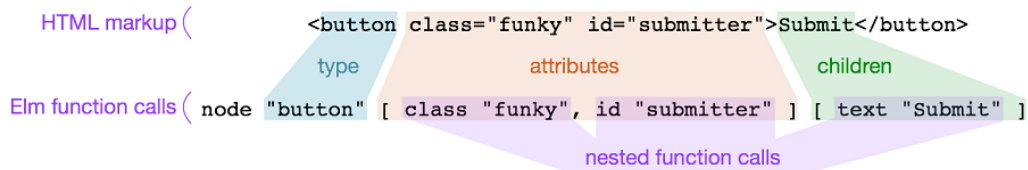


Figure 2.2 Representing a button using HTML markup (top) and Elm's node function (bottom)

There are four function calls in this line of Elm code. Can you spot them?

1. A call to the `node` function, passing three arguments: the string `"button"`, a list of attributes, and a list of child nodes
2. A call to the `class` function, passing `"funky"`
3. A call to the `id` function passing `"submitter"`
4. A call to the `text` function passing `"Submit"`

These are plain old Elm functions. Each returns a representation of some portion of the DOM: a `button` element, a text node, and some `class` and `id` attributes. You can call these functions anytime you like, and pass their return values to other functions as normal.

In Elm we usually refer to a “a virtual DOM node” as “`Html`” for short. This name comes from the `Html` module, which provides a variety of intuitively-named functions that let you avoid calling `node` directly. For example, the `Html` module’s `img` function is shorthand for calling `node` passing `"img"` as the first argument. The following two lines are equivalent:

```
node "img" [ src "logo.png" ] []
img [ src "logo.png" ] []
```

It’s best practice to use functions like `img` as much as possible, and to fall back on `node` only in cases where no alternative is available. (For example, you may notice that there is no equivalent of the `img` function for the deprecated `<blink>` element. I’m not saying you should call `node "blink" [] [text "<BLINK> LIVES AGAIN"]`, but I’m not *not* saying it either.)

RENDERING A PAGE

Let’s use what we’ve learned to render our first page with Elm!

For the rest of this chapter we’ll be building an application called *Photo Groove*. Eventually we’ll add features like searching and viewing larger versions, but first we need to render a basic page that says “Photo Groove” across the top, with some thumbnail images below.

Since our output is visual this time, `elm repl` won’t get us very far. Instead, let’s kick things off with our first `.elm` file!

1. Make a new directory called `PhotoGroove` and open it in a terminal.
2. Run `elm init` and enter `Y` when prompted. This will create a file called `elm.json`, which Elm needs to build our project. We’ll take a closer look at `elm.json` in later chapters.

3. `elm init` will have created a directory called `src`. Create a file called `PhotoGroove.elm` inside this `src` directory.

Now that our file structure is set up, enter the following into the `src/PhotoGroove.elm` file:

Listing 2.2 `src/PhotoGroove.elm`

```
module PhotoGroove exposing (main)                                #A

import Html exposing (div, h1, img, text)                         #B
import Html.Attributes exposing (..)                               #B

view model =
    div [ class "content" ]
        [ h1 [] [ text "Photo Groove" ]                          #C
        , div [ id "thumbnails" ]
            [ img [ src "http://elm-in-action.com/1.jpeg" ] []
              , img [ src "http://elm-in-action.com/2.jpeg" ] []
              , img [ src "http://elm-in-action.com/3.jpeg" ] [] #E
            ]
        ]

main =                                                            #F
    view "no model yet"
```

#A declaring a new module
 #B importing other modules
 #C h1 element with an empty attributes list
 #D Put commas at the start of the line
 #E img element with an empty children list
 #F We'll discuss "main" later

DECLARING THE PHOTOGROOVE MODULE

By writing `module PhotoGroove exposing (main)` at the top of our `PhotoGroove.elm` file, we defined a new module. This means future modules in our project will be able to import this `PhotoGroove` module just like they would the `String` or `Html` modules, for example like so:

```
import PhotoGroove exposing (main)
```

Because we wrote `exposing (main)` after `module PhotoGroove`, we are exposing only one of our top-level values—that is, `main` but not `view`—for other modules to import. This means another module that imported `PhotoGroove` would get an error if it tried to access `PhotoGroove.view`. Only exposed values can be accessed by other modules!

As we will see in later chapters, it's best for our modules to expose as little as possible.

Why Commas in Front?

When writing a multi-line literal in JavaScript, the usual convention is to put commas at the end of each line. Consider the following code:

```
rules = [
  rule("Do not talk about Sandwich Club."),
  rule("Do NOT talk about Sandwich Club.")
  rule("No eating in the common area.")
]
```

Did you spot the mistake? There's a comma missing after the second call to `rule`, meaning this is not syntactically valid JavaScript. Running this code will result in a `SyntaxError`.

Now consider the equivalent Elm code, with the same missing comma:

```
rules = [
  rule "Do not talk about Sandwich Club.",
  rule "Do NOT talk about Sandwich Club."
  rule "No eating in the common area."
]
```

The mistake is just as easy to overlook, but harder to fix because this is syntactically valid Elm code—just not the code you intended to write!

The missing comma means the above code is essentially equivalent to the following:

```
rules = [
  (rule "Do not..."),
  (rule "Do NOT..." rule "No eating...")
]
```

Instead of calling `rule` three times, each time with one argument, here the second call to `rule` is receiving three arguments—and there is no third call! This means instead of the syntax error JavaScript gave you, you'll get a seemingly nonsensical error about functions being called with the wrong number of arguments.

Now try to make this mistake when writing in a commas-first style:

```
rules =
  [ rule "Do not talk about Sandwich Club."
    rule "Do NOT talk about Sandwich Club."
    , rule "No eating in the common area."
  ]
```

This style makes it blindingly obvious that a comma is missing. Now we don't even need to compile our code to identify the problem!

It may feel different at first, but the commas-first style gives you one less potential error to worry about once you get used to it.

Now that we have our `PhotoGroove` module, it's time to see what it looks like in a browser!

Still inside the same directory as the one where we ran `elm init`, run this:

```
elm reactor
```


This will start up a local server running at <http://localhost:8000> which can compile and serve our Elm files. Open <http://localhost:8000/src/PhotoGroove.elm> in your browser, and you should see the results of the compiled `PhotoGroove.elm` file. The page should look like the screenshot in Figure 2.3.

Photo Groove



Figure 2.3 Viewing <http://localhost:8000/src/PhotoGroove.elm>

Congratulations! You've rendered your first user interface in Elm.

Figure 2.4 shows the DOM structure of the interface we just rendered.

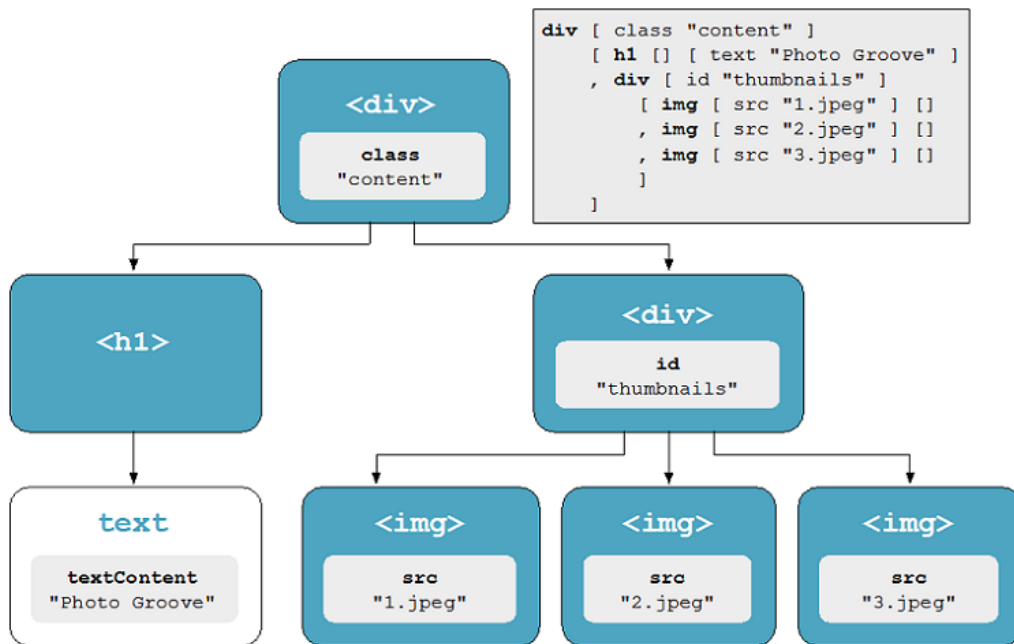


Figure 2.4 The DOM structure of your first Elm user interface

Notice how the functions that create elements—in this case, `div`, `h1`, and `img`—take exactly two arguments in all cases:

1. **A list of attributes.** If an element has no attributes, we pass `[]` like so:
`h1 [] [text "Photo Groove"]`
2. **A list of child DOM nodes.** If an element has no children, we pass `[]` like so:
`img [src "1.jpeg"] []`

If an element has neither attributes nor children? In that case we pass `[] []` like this:

```
br [] []
```

This two-argument pattern is consistent throughout the `Html` module, and it's worth following if you ever decide to make a custom element of your own using the `node` function. We'll see more of the `node` function in Chapter 5, *Talking to JavaScript*.

IMPORTING UNQUALIFIED VALUES WITH 'EXPOSING'

We've now used several functions from the `Html` module, but we wrote them in a different style from the way we did in Chapter 1. Back then we wrote functions like `String.filter` in a *qualified* style—that is, we included the `String` module's name right there in the function call.

With the `Html` module's functions, we used an *unqualified* style—we wrote `div` instead of `Html.div`, we wrote `h1` instead of `Html.h1`, and so forth.

We could do this because we used `exposing` when we imported the `Html` module:

```
import Html exposing (div, h1, img, text)
```

This line of code both imports the `Html` module so we can use its contents, and also brings `Html.div`, `Html.h1`, `Html.img`, and `Html.text` into the global scope. That lets us refer to them as `div`, `h1`, `img`, and `text` without the prefix of `Html`.

We could have achieved essentially the same result by assigning them names directly:

```
import Html

div = Html.div
h1 = Html.h1
img = Html.img
text = Html.text
```

However, since this pile of code can be replaced by a single line—`import Html exposing (div, h1, img, text)`—it's normal to use `exposing` for this purpose instead.

EXPOSING EVERYTHING WITH `(..)`

When we imported the `Html` module, we listed exactly which values we wanted to expose: `div`, `h1`, `img`, and `text`. For the `Html.Attributes` module, we wrote this instead.

```
import Html.Attributes exposing (..)
```

Using `exposing (..)` means “expose everything,” which lets us use every value in the `Html` module in an unqualified style. Let's change our first `import` to use `exposing (..)` instead:

```
import Html exposing (..)
```

Now we won't need to extend the list of `div`, `h1`, `img`, and `text` whenever we want to use a new element type. Everything the `Html` module has to offer is now in our global scope!

Why is the qualified style the better default choice?

In Chapter 1 we wrote out `String.toUpper` and `List.filter`, instead of `toUpper` and `filter`. Here we're doing the opposite, writing `img` and `div` instead of `Html.img` and `Html.div`.

This begs the question: when is it a good idea to use the qualified style (with the module name prefixed) over the unqualified style? The unqualified style is more concise, so why not use `exposing (..)` every time?

There are two primary downsides to unqualified imports. One is that unqualified names can become ambiguous. Try this in `elm repl`:

```
> import String exposing (..)
> import List exposing (..)
> reverse
```

You'll get an error saying that `reverse` is ambiguous. After importing and exposing both `String.reverse` and `List.reverse`, it's no longer clear which of the two you meant! (In cases like this you can still use the qualified style to resolve the ambiguity, so if you now put `String.reverse` or `List.reverse` into `elm repl`, they will still work as normal.)

Unqualified imports are also less self-documenting. Suppose you come across code that says `partition foo bar`, and you've never seen `partition` before. Naturally you wonder: "How can I find out what `partition` does? Is it defined in this file?" You search through the file and can't find it, so it must come from an `import`. You scroll up to the imports and discover a long list of `exposing (..)` declarations. Argh! `partition` could be in any of those!

This could take awhile...

Suppose instead you see the code `List.partition foo bar`. You want to know what `List.partition` does, so you bring up the documentation for the `List` module on package.elm-lang.org. You learn about `List.partition`, then get on with your day!

Scenarios like this are why it's best practice to write things in a *qualified* way by default.

Still, sometimes there's a good reason to prefer the unqualified style—like how unqualified `Html` functions are designed to resemble HTML markup. In these cases, it's best to limit yourself to one `exposing (..)` (or perhaps one "family" of them, such as `Html` and `Html.Attributes`) per file. This way if you encounter an unfamiliar function of mysterious origin, you'll have the fewest modules to hunt through to find its documentation!

2.1.2 Building a Project

Now that we've gotten something on the screen, let's add some styles!

TIP There are many ways to style a Web page, each with its own tradeoffs. The two most popular Elm-specific choices are `rtfeldman/elm-css`, a package for writing CSS directly in Elm, and the groundbreaking `mdgriffith/elm-ui`, which provides a way to style pages without writing CSS at all. Comparing styling alternatives is outside the scope of this book, but both of these can be found on package.elm-lang.org.

We could style our page using a separate Elm package, or by writing inline CSS styles using the `Html.Attributes.style` attribute, but instead we're going to organize things by writing our CSS declarations in a separate `.css` file.

The only way to get multiple files involved in the same Web page is to give a browser some HTML markup, so our first step in the process of styling our application will be to create a `.html` file.

Let's a file named `index.html` in the same directory as our `elm.json` file (the one where we ran `elm init` back in section 2.2.1.) and put these contents inside it:

Listing 2.4 index.html

```
<!doctype html>
<html>
  <head>
    <style>
      body { background-color: rgb(44, 44, 44); color: white; }
      img { border: 1px solid white; margin: 5px; }
      .large { width: 500px; float: right; }
```

```

        .selected { margin: 0; border: 6px solid #60b5cc; }
        .content { margin: 40px auto; width: 960px; }
        #thumbnails { width: 440px; float: left }
        h1 { font-family: Verdana; color: #60b5cc; }
    </style>
</head>

<body>
    <div id="app"></div> #A

    <script src="elm.js"></script> #B
    <script>
        Elm.PhotoGroove.init({node: document.getElementById("app")}); #C
    </script>
</body>
</html>

```

#A Our Elm application will render into this div
 #B PhotoGroove.elm will get compiled into elm.js
 #C The Elm object comes from elm.js

The markup we put in this file covers things like:

- The standard `<!doctype>`, `<html>`, and `<body>` tags
- Whatever `<head>` inclusions we need—styles, metadata, `<title>`, and so on
- Importing a file called `elm.js`, which we will have Elm’s compiler generate in a moment

The line `Elm.PhotoGroove.init({node: document.getElementById("app")});` starts our Elm code running in the `<div id="elm-area"></div>` element we included in `index.html`.

COMPILING TO JAVASCRIPT

Next it’s time to compile our Elm code into JavaScript. Run this in the terminal:

```
elm make src/PhotoGroove.elm --output elm.js
```

This will compile our `PhotoGroove.elm` file into the JavaScript file the browser will read. (That generated JavaScript file will be called `elm.js`, because we passed `--output elm.js` to `elm make`.) Now our HTML file has a compiled `elm.js` file to load up!

THE ELM RUNTIME AND ‘MAIN’

When Elm compiles our code into JavaScript, it includes an extra bit of JavaScript known as the Elm Runtime. The Elm Runtime is behind-the-scenes code that quietly handles things like:

- Adding and removing event listeners for any events our code depends on
- Efficiently scheduling tasks like HTTP requests and DOM updates
- Storing and managing application state

When we called `Elm.PhotoGroove.init` from `index.html`, we told the Elm Runtime to use the top-level `main` value in the `PhotoGroove` module as the application’s entry point. If we did not

have a module called `PhotoGroove`, or if it did not define a top-level value named `main`, we'd have gotten an error.

This means when the browser runs our compiled code, `view "no model yet"` will be the first line of code executed, because that's what we assigned to `main`. If we renamed the `PhotoGroove` module to `CubeDraft`, we'd have to call `Elm.CubeDraft.init` instead, but otherwise everything would still work. If the `CubeDraft` module did not define a value named `main`, however, the application would not start. There's no renaming `main`!

If you open `index.html`, you should see the application displaying as it does in Figure 2.5.

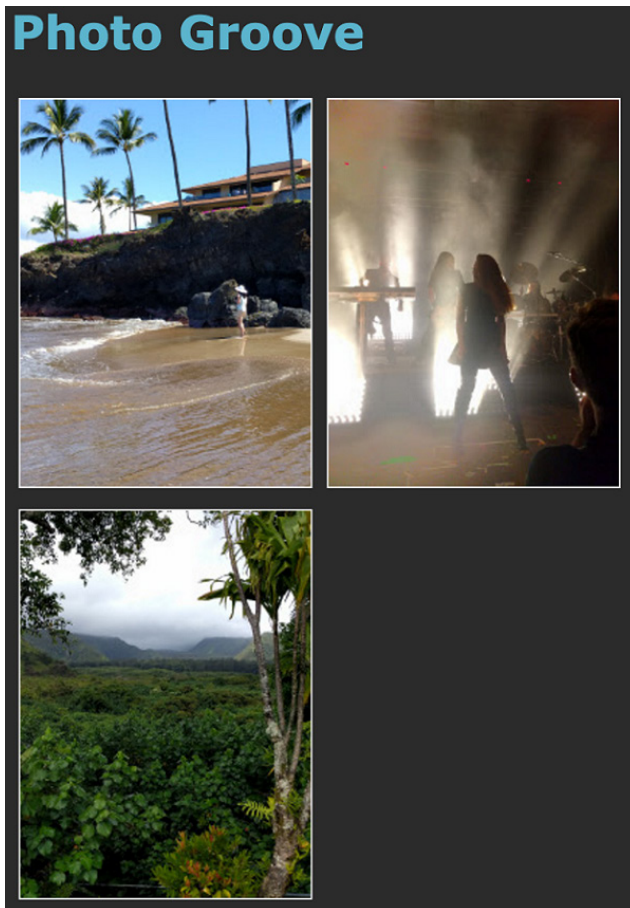


Figure 2.5 Rendering the application

Fantastic! Next we'll make it interactive.

2.2 Handling User Input with the Elm Architecture

So far we haven't had much data flowing through our application. Okay, really we haven't had *any*—all we did was generate some `Html` and render it. That will soon change, as we're about to start handling user input! This brings us to a common question that every growing application faces sooner or later: how will we keep data flow manageable as our code scales?

JavaScript offers a staggering selection of data flow architectures to choose from, but Elm has just one. It's called the Elm Architecture, and the Elm Runtime is optimized for applications that follow it. We'll learn about the Elm Architecture as we add interactivity to Photo Groove.

Figure 2.6 shows a preview of the architecture we'll be building toward in this chapter. Don't worry if this does not make sense yet! We will get there, one step at a time.

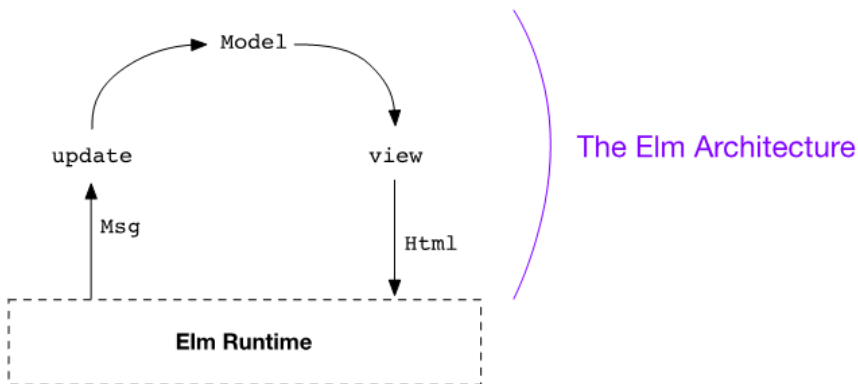


Figure 2.6 The Elm Runtime uses the Elm Architecture to manage data flow

Let's begin where data flow naturally begins in an application: with the application's *state*.

2.2.1 Representing Application State with a Model

Back in the Wild West days of the Web, it was common to store application state primarily in the DOM itself. Is that menu expanded or collapsed? Check whether one of its DOM nodes has `class="expanded"` or `class="collapsed"`. Need to know what value a user has selected in a dropdown? Query it out of the DOM at the last possible instant.

This approach turned out not to scale very well, especially as applications grew more complex. Today it's common practice to store application state completely outside the DOM, and to propagate changes from that independent state over to the DOM as necessary. This is how we do it in the Elm Architecture.

DECLARING A MODEL

We're going to store our application state separately from the DOM, and we'll refer to that state as our *model*.

DEFINITION The *model* represents the state of an Elm application.

Remember how earlier we wrote this code?

```
main =
    view "no model yet"
```

Let's replace this code with the contents of Listing 2.5.

Listing 2.5 Adding a Model

```
initialModel =
    [ { url = "1.jpeg" } #A
    , { url = "2.jpeg" } #A
    , { url = "3.jpeg" } #A
    ]
```

```
main =
    view initialModel #B
```

#A We'll add more fields beyond url later

#B Pass our new initialModel record to view

Excellent! Now we have an initial model to work with. So far it contains a list of photos, each of which is represented by a record containing a `url` string.

WRITING A VIEW FUNCTION

Next we'll render a thumbnail for each photo in our list.

At the top level of a typical Elm application is a single view function, which accepts our current model as an argument and then returns some `Html`. The Elm Runtime takes the `Html` returned by this view function and alters the page's actual DOM to match it.

By pure coincidence, we've already written just such a view function—it's the function we had the foresight to name `view`. Unfortunately, our current `view` implementation ignores the `model` argument it receives, which means changing our model won't result in a visible change to the end user. Let's fix that! `view` should base its return value on its `model` argument.

It'll be easier to do this if we first write a separate `viewThumbnail` *helper function*, which renders a single thumbnail as `Html`.

DEFINITION A *helper function* helps another function do its job. Here, the `viewThumbnail` helper function will help `view` do its job.

Let's replace our `view` implementation with the following:

Listing 2.6 Splitting out `viewThumbnail`

```
urlPrefix = #A
    "http://elm-in-action.com/" #A

view model =
    div [ class "content" ]
      [ h1 [] [ text "Photo Groove" ]
        , div [ id "thumbnails" ] []
      ]

viewThumbnail thumbnail =
    img [ src (urlPrefix ++ thumbnail.url) ] [] #B
```

#A We'll prepend this to strings like "1.jpeg"

#B Prepend `urlPrefix` to get a complete URL like "http://elm-in-action.com/1.jpeg"

Figure 2.7 illustrates how our current `model` and `view` connect to the Elm Runtime.

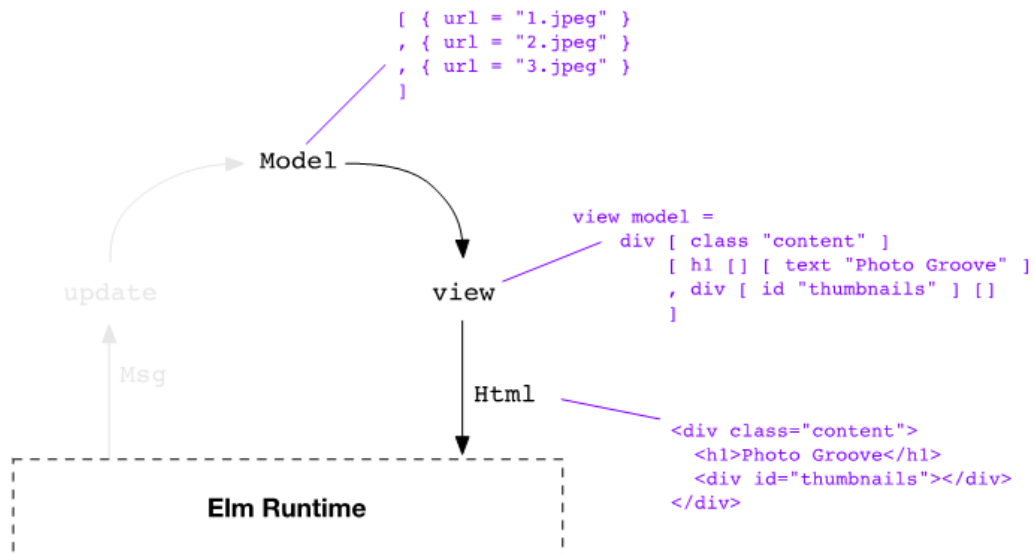


Figure 2.7 Model and view connecting with the Elm Runtime

Next, we'll iterate over our list of photo records and call `viewThumbnail` on each one, in order to translate it from a dusty old record to a vibrant and inspiring `img`.

Fortunately, the `List.map` function does exactly this!

LIST.MAP

`List.map` is another *higher-order* function similar to the `List.filter` function we used in Chapter 1. We pass `List.map` a translation function and a list, and it runs that translation function on each value in the list. Once that's done, `List.map` returns a new list containing the translated values.

Take a look at Figure 2.8 to see `List.map` do its thing for `viewThumbnail`.

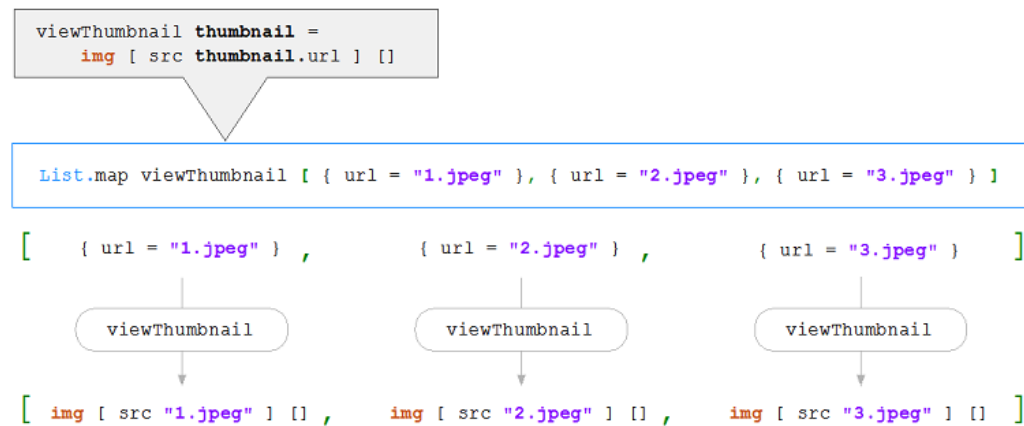


Figure 2.8 Using `List.map` to transform photo records into `img` nodes

Since `div` is just a plain Elm function that accepts two lists as arguments—first a list of attributes, followed by a list of child nodes—we can swap out our entire hardcoded list of child `img` nodes with a single call to `List.map`! Let's do that now.

```
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
      , div [ id "thumbnails" ] (List.map viewThumbnail model)
    ]

viewThumbnail thumbnail =
  img [ src (urlPrefix ++ thumbnail.url) ] []
```

If you run `elm make src/PhotoGroove.elm --output elm.js` again to recompile this code, you should see the same result as before. The difference is that now we have a more flexible internal representation, setting us up to add interactivity in a way that was impossible before we connected `model` and `view`.

EXPANDING THE MODEL

Now let's add a feature: when the user clicks on a thumbnail, it will become selected—indicated by a blue border surrounding it—and we'll display a larger version of it beside the thumbnails.

To do this, we first need to store which thumbnail is selected. That means we'll want to convert our model from a list to a record, so we can store both the list of photos and the current `selectedUrl` value at the same time.

Listing 2.7 Converting the model to a record

```
initialModel =
  { photos =
    [ { url = "1.jpeg" }
    , { url = "2.jpeg" }
    , { url = "3.jpeg" }
    ]
  , selectedUrl = "1.jpeg"  #A
  }
```

#A Select the first photo by default

Next let's update `viewThumbnail` to display the blue border for the selected thumbnail.

That's easier said than done! `viewThumbnail` accepts only one argument—`thumbnail`—so it has no way to access the model. That in turn means it can't possibly know the current value of `selectedUrl`...but without knowing which thumbnail is selected, how can it know whether to return a selected or unselected `img`?

It can't! We'll have to pass that information along from `view` to `viewThumbnail`.

Let's rectify this situation by passing `selectedUrl` into `viewThumbnail` as an additional argument. Armed with that knowledge, it can situationally return an `img` with the "selected" class—which our CSS has already styled to display with a blue border—if the `url` of the given `thumbnail` matches `selectedUrl`.

```
viewThumbnail selectedUrl thumbnail =
  if selectedUrl == thumbnail.url then
    img
      [ src (urlPrefix ++ thumbnail.url)
      , class "selected"
      ]
    []
  else
    img
      [ src (urlPrefix ++ thumbnail.url) ]
    []
```

Comparing our `then` and `else` cases, we see quite a bit of code duplication. The only thing different about them is whether `class "selected"` is present. Can we trim down this code?

Absolutely! We can use the `Html.classList` function. It builds a `class` attribute using a list of tuples, with each tuple containing first the desired class name, and second a boolean for whether to include the class included in the final class string.

Let's refactor our above code to the following, which accomplishes the same thing:

```
viewThumbnail selectedUrl thumbnail =
  img
    [ src (urlPrefix ++ thumbnail.url)
    , classList [ ( "selected", selectedUrl == thumbnail.url ) ]
    ]
  []
```

Now all that remains is to pass in `selectedUrl`, which we can do with an anonymous function. While we're at it, let's also add another `img` to display a larger version of the selected photo.

Listing 2.8 Rendering Selected Thumbnail via anonymous function

```
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
    , div [ id "thumbnails" ]
      (List.map (\photo -> viewThumbnail model.selectedUrl photo)
        model.photos
      )
    , img
      [ class "large" #A
      , src (urlPrefix ++ "large/" ++ model.selectedUrl)
      ]
    []
  ]
```

#A Display a larger version of the selected photo

If you recompile with the same `elm make` command as before, the result should now look like Figure 2.9.

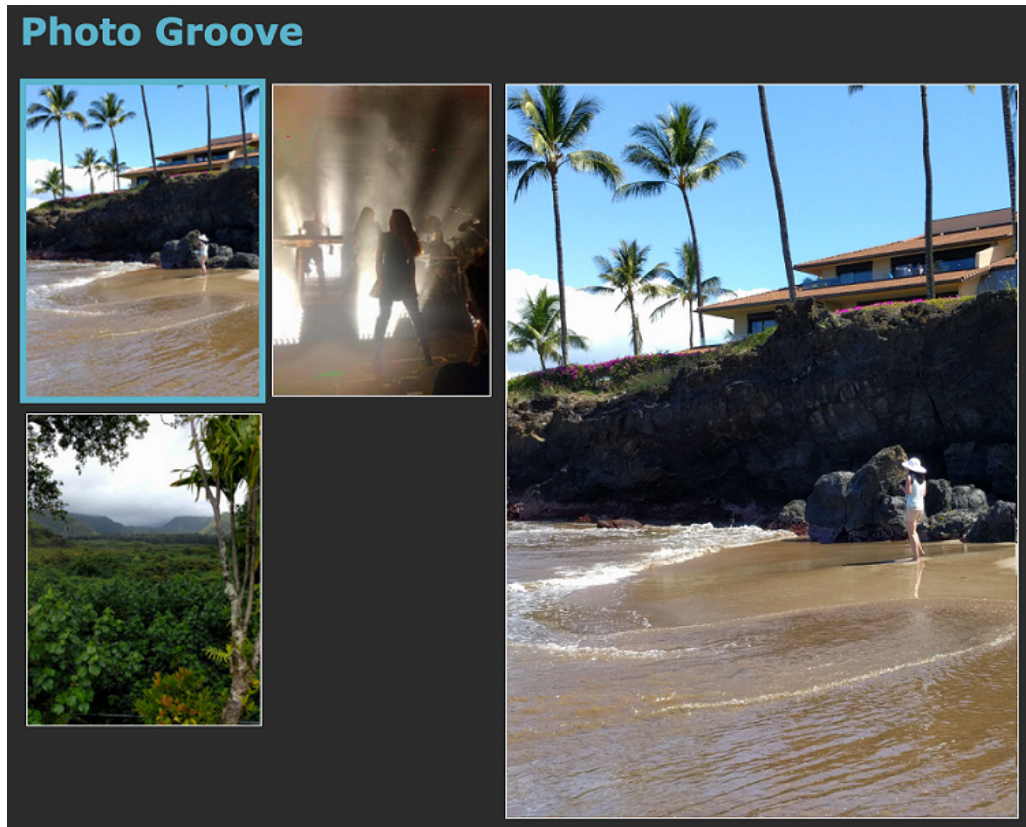


Figure 2.9 Rendering the selected thumbnail alongside a larger version.

Looking good!

REPLACING ANONYMOUS FUNCTIONS WITH PARTIAL APPLICATION

Although the way we've written this works, it's not quite idiomatic Elm code. The idiomatic style would be to remove the anonymous function like so:

```
Before: List.map (\photo -> viewThumbnail model.selectedUrl photo) model.photos
After: List.map (viewThumbnail model.selectedUrl) model.photos
```

Whoa! Does the revised version still work? Do these two lines somehow do the same thing?

It totally does, and they totally do! This is because calling `viewThumbnail` without passing all of its arguments is an example of *partially applying* a function.

DEFINITION *Partially applying* a function means passing it some of its arguments—but not all of them—and getting back a new function which will accept the remaining arguments and finish the job.

When we called `viewThumbnail model.selectedUrl photo`, we provided `viewThumbnail` with both of the arguments it needed to return some `Html`. If we call it without that second `photo` argument, what we get back is not `Html`, but rather a function—specifically a function that accepts the missing `photo` argument and *then* returns some `Html`.

Let’s think about how this would look in JavaScript, where functions don’t support partial application by default. If we’d written `viewThumbnail` in JavaScript, and wanted it to support partial application, it would have had to look like this:

```
function viewThumbnail(selectedUrl) {
  return function(thumbnail) {
    if (selectedUrl === thumbnail.url) {
      // Render a selected thumbnail here
    } else {
      // Render a non-selected thumbnail here
    }
  };
}
```

Functions that can be partially applied, such as the one in this JavaScript code, are known as *curried* functions.

DEFINITION A *curried* function is a function that can be partially applied.

All Elm functions are curried. That’s why when we call `(viewThumbnail model.selectedUrl)` we end up partially applying `viewThumbnail`, not getting an `undefined` argument or an error.

In contrast, JavaScript functions are not curried by default. They are instead *tupled*, which is to say they expect a complete “tuple” of arguments. (In this case, “tuple” refers to “a fixed-length sequence of elements,” not specifically one of Elm’s `Tuple` values.)

Elm and JavaScript both support either curried or tupled functions. The difference is which they choose as the default:

- In JavaScript, functions are tupled by default. If you’d like them to support partial application, you can first curry them by hand—like we did in our JavaScript `viewThumbnail` implementation above.
- In Elm, functions are curried by default. If you’d like to partially apply them...go right ahead! They’re already set up for it. If you’d like a tupled function, write a curried function that accepts a single `Tuple` as its argument, then destructure that tuple.

Table 2.1 shows how to define and use both curried and tupled functions in either language.

Table 2.1 Curried functions and Tupled functions in Elm and JavaScript

	Elm	JavaScript
Curried Function	<pre>splitA separator str = String.split separator str</pre>	<pre>function splitA(sep) { return function(str) { return str.split(sep); } }</pre>
Tupled Function	<pre>splitB (separator, str) = String.split separator str</pre>	<pre>function splitB(sep, str) { return str.split(sep); }</pre>
Total Application	<pre>splitB ("-", "867-5309")</pre>	<pre>splitB("-", "867-5309")</pre>
Total Application	<pre>splitA "-" "867-5309"</pre>	<pre>splitA("-")("867-5309")</pre>
Partial Application	<pre>splitA "-"</pre>	<pre>splitA("-")</pre>

We can use our newfound powers of partial application to make `view` more concise! We now know we can replace our anonymous function with a partial application of `viewThumbnail`.

Before: `List.map (\photo -> viewThumbnail model.selectedUrl photo) model.photos`
After: `List.map (viewThumbnail model.selectedUrl) model.photos`

TIP In Elm, an anonymous function like `(\foo -> bar baz foo)` can always be rewritten as `(bar baz)` by itself. Keep an eye out for this pattern; it comes up surprisingly often.

Here's how our updated `view` function should look.

Listing 2.9 Rendering Selected Thumbnail via partial application

```
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
    , div [ id "thumbnails" ]
      (List.map (viewThumbnail model.selectedUrl) model.photos) #A
    , img
      [ class "large"
      , src (urlPrefix ++ "large/" ++ model.selectedUrl)
      ]
      []
    ]
```

#A Partially apply `viewThumbnail` with `model.selectedUrl`

Since all Elm functions are curried, it's common to give a function more information by adding an argument to the **front** of its arguments list.

For example, when `viewThumbnail` needed access to `selectedUrl`, we made this change:

```
Before: List.map viewThumbnail model.photos
After:  List.map (viewThumbnail model.selectedUrl) model.photos
```

Because we added the new `selectedUrl` argument to the front, we could pass it in using partial application instead of an anonymous function. This is a common technique in Elm code!

NOTE Currying is named after acclaimed logician Haskell Brooks Curry. The Haskell programming language is also named after his first name. Whether the Brooks Brothers clothing company is named after his middle name is left as an exercise to the reader.

2.2.2 Handling Events with Messages and Updates

Now that we can properly render which thumbnail is selected, we need to change the appropriate part of the model whenever the user clicks a different thumbnail.

If we were writing JavaScript, we might implement this logic by attaching an event listener to each thumbnail like so:

```
thumbnail.addEventListener("click", function() { model.selectedUrl = url; });
```

Elm wires up event handlers a bit differently. Similarly to how we wrote a `view` function that translated our current `model` into a desired DOM structure, we're now going to write an `update` function that translates *messages* into our desired `model`.

DEFINITION A *message* is a value used to pass information from one part of the system to another.

When the user clicks a thumbnail, a message will be sent to an `update` function as follows:



Figure 2.10 Handling the event when a user clicks a thumbnail

The message should describe **what happened**—for example, “the user clicked a photo”—but the format of our message is entirely up to us. We could represent it as a string, or a list, or a number, or anything else we please. Here’s a message implemented as a record:

```
{ description = "ClickedPhoto", data = "2.jpeg" }
```

This record is a message which conveys the following information:

“The user clicked the 2.jpeg photo.”

It will be up to our `update` function to decide what to do with this information. In general, when `update` receives a message it will do the following:

1. Look at the message it received.
2. Look at our current model.
3. Use these two values to determine a new model, then return it.

We can implement our “select photo” logic by adding this `update` function right above `main`:

```
update msg model =
  if msg.description == "ClickedPhoto" then
    { model | selectedUrl = msg.data }

  else
    model
```

Notice how if we receive an unrecognized message, we return the original model unchanged. This is important! Whatever else happens, the `update` function must always return a new model, even if it happens to be the same as the old model.

ADDING ONCLICK TO VIEWTHUMBNAIL

We can declare that a `ClickedPhoto` message should be sent to `update` whenever the user clicks a thumbnail, by adding an `onClick` attribute to `viewThumbnail`. To do this, we’ll first need to import the `Html.Events` module, since that’s where `onClick` lives!

While we’re at it, let’s change `Html.Attributes` to expose `(..)`, such that our imports now look like this:

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onClick)
```

Now that we’ve imported `onClick`, let’s introduce it to `viewThumbnail` like so:

```
viewThumbnail selectedUrl thumbnail =
  img
    [ src (urlPrefix ++ thumbnail.url)
    , classList [ ( "selected", selectedUrl == thumbnail.url ) ]
    , onClick { description = "ClickedPhoto", data = thumbnail.url }
    ]
  []
```

The Elm Runtime takes care of managing event listeners behind the scenes, so this one-line addition is the only change we need to make to our view. We're ready to see this in action!

THE MODEL-VIEW-UPDATE LOOP

To wire our Elm application together, we're going to change `main = view model` to the following, which incorporates `update` according to how we've set things up so far. First we'll add one final import to the top of our imports list:

```
import Browser
```

This module gives us access to the `Browser.sandbox` function, which we can use to describe an interactive Elm application like so:

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

The `Browser.sandbox` function takes a record with three fields:

- **model** - A value that can be anything you please.
- **view** - A function that takes a model and returns a `Html` node.
- **update** - A function that takes a message and a model, and returns a new model.

It uses these arguments to return a description of a program, which the Elm Runtime sets in motion when the application starts up. Before we got the `Browser` module involved, `main` could only render static views. `Browser.sandbox` lets us specify how to react to user input!

Figure 2.11 demonstrates how data flows through our revised application.

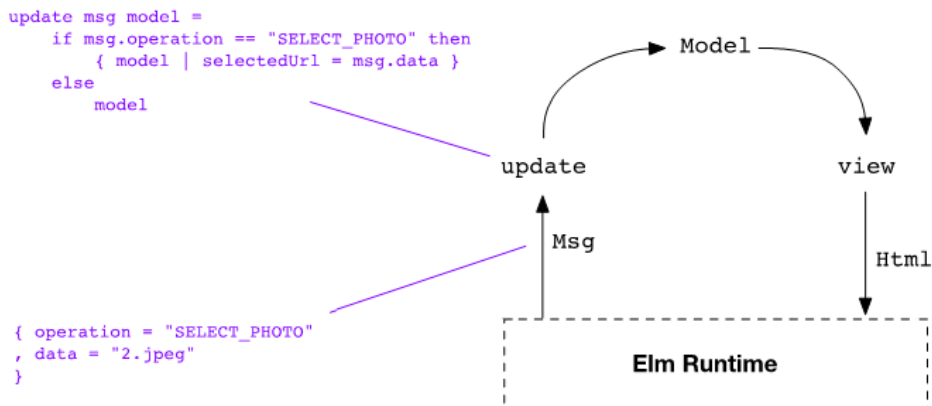


Figure 2.11 Data flowing from the start of the program through the Model-View-Update loop

Notice that `view` builds fresh `Html` values after every `update`. That might sound like a lot of performance overhead, but in practice, it's almost always a performance *benefit*!

This is because Elm doesn't actually recreate the entire DOM structure of the page every time. Instead, it compares the `Html` it got this time to the `Html` it got last time and updates only the parts of the page that are different between the two requested representations.

This approach to "Virtual DOM" rendering, popularized by the JavaScript library React, has several benefits over manually altering individual parts of the DOM:

- Updates are automatically batched to avoid expensive repaints and layout reflows
- It becomes far less likely that application state will get out of sync with the page
- Replaying application state changes effectively replays user interface changes

Let's compile once more with `elm make src/PhotoGroove.elm --output elm.js`. If you open `index.html`, you should now be able to click a thumbnail to select it. Huzzah!

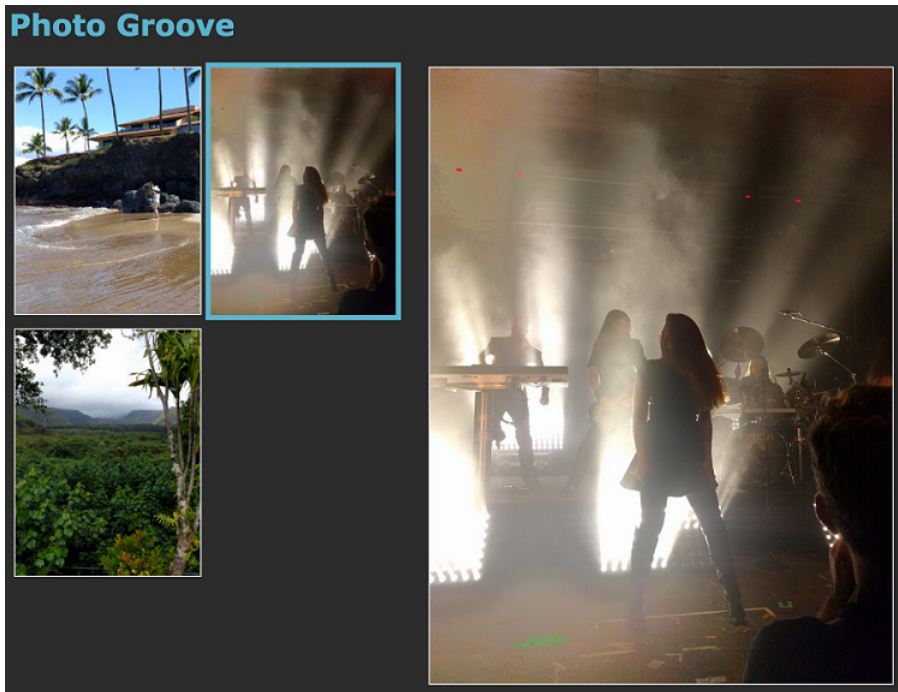


Figure 2.12 Our final Photo Groove application

At this point we've also worked our way through the complete Elm Architecture diagram from the beginning of the chapter. Figure 2.13 shows where things ended up.

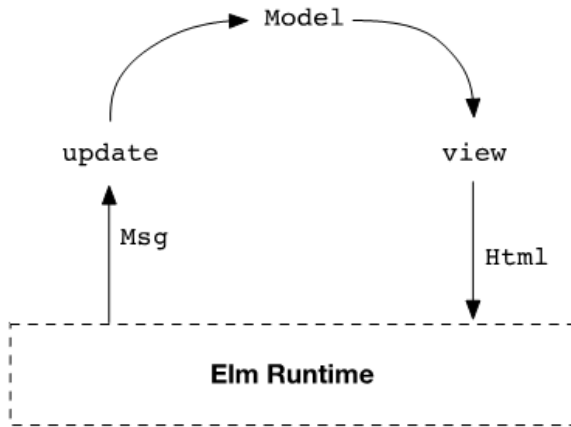


Figure 2.13 Our final Elm Architecture setup

Congratulations on a job well done!

2.3 Summary

In this chapter we learned about three ways to handle interactions, each of which differs from the JavaScript way of handling the same. Table 2.2 summarizes these differences.

Table 2.2 Handling interactions in JavaScript compared to Elm

Interaction	JavaScript approach	Elm approach
Changing the DOM	Directly alter DOM nodes	Return some <code>Html</code> from a view function
Reacting to user input	Attach a listener to an element	Specify a message to send to update
Changing application state	Alter an object in place	Return a new model in update

We covered many other concepts in the course of building our first Elm application, including:

- A model represents our application state
- A view function takes a model and returns a list of `Html` nodes
- User events such as clicks get translated into message values
- Messages get run through the `update` function to produce a new model
- After an `update`, the new model is sent to the view function to determine the new DOM
- `Html.beginnerProgram` wires together `model`, `view`, and `update`
- `List.map` is a higher-order function that translates one list into another
- All Elm functions are curried, which means they can be partially applied

Here's the complete `src/PhotoGroove.elm` file we ended up with at the end:

Listing 2.10 PhotoGroove.elm with complete Model-View-Update in place

```

module PhotoGroove exposing (main)                                #A

import Browser                                                    #B
import Html exposing (..)                                          #B
import Html.Attributes exposing (..)                               #B
import Html.Events exposing (onClick)                             #B

urlPrefix =
    "http://elm-in-action.com/"

view model =                                                      #C
    div [ class "content" ]
        [ h1 [] [ text "Photo Groove" ]
        , div [ id "thumbnails" ]
            (List.map (viewThumbnail model.selectedUrl) model.photos) #D
        , img
            [ class "large"
            , src (urlPrefix ++ "large/" ++ model.selectedUrl)
            ]
            []
        ]

viewThumbnail selectedUrl thumbnail =
    img
        [ src (urlPrefix ++ thumbnail.url)
        , classList [ ( "selected", selectedUrl == thumbnail.url ) ]
        , onClick { description = "ClickedPhoto", data = thumbnail.url } #E
        ]
        []

initialModel =
    { photos =
        [ { url = "1.jpeg" }
        , { url = "2.jpeg" }
        , { url = "3.jpeg" }
        ]
    , selectedUrl = "1.jpeg"
    }

update msg model =
    if msg.description == "ClickedPhoto" then
        { model | selectedUrl = msg.data } #F
    else
        model

main =
    Browser.sandbox #G
        { init = initialModel
        , view = view
        }

```

```
, update = update  
}
```

#A The name of our module
#B The other modules we're importing
#C The view function takes the current model and returns some Html
#D viewThumbnail is partially applied here
#E When the user clicks, this message is sent to update
#F Change the selected URL to the photo the user clicked
#G Browser.sandbox describes our complete application

In the next chapter we'll get into ways to improve upon the application we've made so far, both adding features and refactoring to make it easier to maintain.

Onward!

3

Compiler as Assistant

This chapter covers

- Documenting guarantees with type annotations
- Implementing multi-way conditionals with *case-expressions*
- Storing flexible data with *custom types*
- Using `Array` and `Maybe` for positional element access
- Generating random numbers using *commands*

In Chapter 2 we built our first Elm application. It doesn't do much yet, but it has potential! So far it displays thumbnails of three photos, and lets users click one to view a larger version.

We showed it to our manager, who was thrilled with what we'd made. "Wow, this is looking *incredible*. The part where you click the thumbnail and it shows the bigger version? Just brilliant. I'm going to get some more team members working with you on this."

Nice! Quite a vote of confidence. Sure, we have no tests or documentation to help get these new teammates up to speed, but there's no time like the present to clean up our code!

Our manager also has a couple of feature requests. "Let's give users the ability to choose between viewing small, medium, or large thumbnails. Also, for this next version I want to kick the fun factor into overdrive. Let's add a button that says *Surprise Me!* and when you click it, it selects one of the photos...**at random.**"

We resist the urge to tell our manager "whoa there, that might be too much fun" and instead review the tasks we've just received:

1. Improve code quality to help new team members get up to speed
2. Let users choose between small, medium, and large thumbnails
3. Add a *Surprise Me!* button that randomly selects a photo

Improving code quality *while* adding new features is often a tall order. Even more so because these particular features involve aspects of Elm we have not yet encountered.

Fortunately, we have an assistant to help us out: Elm’s compiler. In this chapter we’ll learn how it can help us improve documentation, refactor without introducing regressions, and accomplish both while introducing new features. Let’s get to it!

3.1 Documenting Guarantees with Type Annotations

One of the quickest ways we can make our code nicer for incoming teammates is to add some comments that document what our code does. Although comments are simple and flexible, they’re also notoriously unreliable. Sometimes they’re written inaccurately. Other times they start out accurate but become inaccurate as the code base changes out from under them.

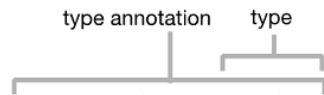
“Code never lies. Comments sometimes do.”

—Ron Jeffries

Elm gives us access to a genuinely trustworthy form of documentation: *type annotations*.

3.1.1 Adding Optional Type Annotations

In Chapter 2 we assigned the constant `urlPrefix` to `"http://elm-in-action.com"`. Let’s edit our `PhotoGroove.elm` file to add a type annotation on top of that `urlPrefix` assignment:



```
urlPrefix : String
urlPrefix =
    "http://elm-in-action.com/"
```

Figure 3.1 Adding a type annotation to `urlPrefix`

This annotation is saying “`urlPrefix` is a `String`,” and it’s not kidding around with that. Elm’s compiler will check our entire code base to verify this claim. If finds even one incident where `urlPrefix` is not used as a `String`, it will give us an error at compile time. This means if our code compiles, we can be certain `urlPrefix` is a `String` absolutely everywhere it’s used!

TIP Searching for `"urlPrefix :"` in your editor is now a quick way to jump to `urlPrefix`’s definition.

This guarantee is even more useful for annotated functions. When teammates are new to the codebase, being able to reliably tell at a glance what arguments a function takes, and what it returns, can be an incredible time-saver.

ANNOTATING FUNCTIONS

We can annotate functions by writing `->` between their arguments and return values.

```
isEmpty : String -> Bool
isEmpty str = str == ""
```

This annotation says “`isEmpty` takes a `String` and returns a `Bool`.” `Bool` refers to one of those `True` or `False` boolean values that we can use as the condition of an *if-expression*.

ANNOTATING RECORDS

Record annotations use `:` instead of `=` but otherwise look about the same as record values.

```
selectPhoto : { description : String, data : String }
selectPhoto = { description = "ClickedPhoto", data = "1.jpeg" }
```

This annotation says “`selectPhoto` is a record with an `description` field and a `data` field, and each of those fields is a `String`.” We can add a similar annotation to our `initialModel`:

```
initialModel =
  { photos = [ { url = "1.jpeg" }, { url = "2.jpeg" }, { url = "3.jpeg" } ]
    , selectedUrl = "1.jpeg"
  }
```

Ah, but this record has a list in it! How do we annotate lists?

ANNOTATING LISTS

In Chapter 1 we learned that all elements in an Elm list must have a consistent type. This is reflected in their type annotations. For example, Elm represents a “list of strings” with the type annotation `List String`, and a “list of booleans” with the type annotation `List Bool`.

Let’s play around with some different lists to see their type annotations in `elm repl`:

Listing 3.1 Annotating Lists

```
> [ "funk", "soul" ]
["funk", "soul"] : List String    #A

> [ [ "don't", "forget" ], [ "about", "Dre" ] ]
[[ "don't", "forget" ], [ "about", "Dre" ]] : List (List String) #B

> [ { url = "1.jpeg" }, { url = "2.jpeg" } ] #C
[{ url = "1.jpeg" }, { url = "2.jpeg" }] : List { url : String }
```

#A “a list of strings”

#B “a list of lists of strings”

#C “a list of records”

Notice anything about the structure of that last example? It looks just like the list of photos in our model! This means we can use it to write an annotation for our model. Let’s do that:

```
initialModel : { photos : List { url : String }, selectedUrl : String }
initialModel =
  ...
```

Model annotations are among the most helpful forms of documentation for new teammates, because they concisely (and reliably!) describe the structure of our entire application state.

3.1.2 Annotating Functions with Type Variables

Functions, records, and lists served us well in Chapter 2, but implementing the *Surprise Me!* button will involve a collection we haven't used before.

LISTS AND ARRAYS

We mentioned in Chapter 1 that Elm supports arrays as well as lists. Both are sequential collections of varying length, whose elements share a consistent type. However, there are a few differences between arrays and lists:

- Lists can be created with square bracket literals, whereas arrays have no literal syntax in Elm. We always create arrays by calling functions.
- Lists have better performance in typical Elm use cases. This makes them the standard choice for representing a sequence of values, in contrast to JavaScript where arrays are standard. In Chapter 10 we'll see why lists are a better standard choice in Elm.
- Arrays are better for arbitrary positional access. That means it's quick and easy to say "give me the 3rd element in this array." Lists have no first-class way to do this.

Randomly choosing a photo involves arbitrary positional access. As such, we've found ourselves in exactly the sort of situation where an array will serve us better than a list!

ARRAY.FROMLIST

The most common way to obtain an array is by calling `Array.fromList` on an existing list. Table 3.1 shows the results of a few different calls to `Array.fromList`.

Table 3.1 Translating a list into an array using `Array.fromList`

Expression	Result
<code>Array.fromList [2, 4, 6]</code>	Array containing 3 elements: 2, 4, and 6
<code>Array.fromList ["foo"]</code>	Array containing 1 element: "foo"
<code>Array.fromList []</code>	Array containing 0 elements

TYPE VARIABLES

How would we write a type annotation for `Array.fromList`? It's bit different from the functions we've annotated so far, as it returns a different type depending on what you pass it.

Table 3.2 shows some examples of the different types of values it can return.

Table 3.2: `Array.fromList`'s return type depends on its input

Given one of these	
List <code>Float</code>	Array <code>Float</code>
List <code>String</code>	Array <code>String</code>
List { <code>url : String</code> }	Array { <code>url : String</code> }

Notice the pattern? Whatever type of list we pass in, that's what type of array we get out.

We can capture this pattern in a type annotate like so:

```
fromList : List elementType -> Array elementType
```

In this annotation, `elementType` is a *type variable*.

DEFINITION A *type variable* represents more than one possible type. Type variables have lowercase names, making them easy to tell them apart from *concrete types* like `String`, which are always capitalized.

When you call `Array.fromList` passing some list, the type variable `elementType` gets replaced by that list's element type. Table 3.3 shows this process in action.

Table 3.3 Replacing `Array.fromList`'s type variable

When we pass this in...	...elementType becomes...	...and we get this back
List <code>Float</code>	<code>Float</code>	Array <code>Float</code>
List <code>String</code>	<code>String</code>	Array <code>String</code>
List { <code>url : String</code> }	{ <code>url : String</code> }	Array { <code>url : String</code> }

CHOOSING NAMES FOR TYPE VARIABLES

When annotating `fromList`, we can choose a different name for its type variable besides `elementType`. Since a type variable essentially serves as a placeholder for a concrete type like `String` or `List Int`, it can have just about any name, so long as it's lowercase and we use it consistently. For example, we could annotate `Array.fromList` in any of the following ways:

- List `elementType` -> Array `elementType`
- List `foo` -> Array `foo`
- List `a` -> Array `a`

In contrast, **none** of the following would work as annotations for `Array.fromList`:

- List `elementType` -> Array `blah`

- `List foo` `-> Array bar`
- `List a` `-> Array b`

These three annotations are saying “`Array.fromList` takes a list containing elements of one type and returns an array that potentially contains elements of another type.”

That just ain’t so! `Array.fromList` can only return an array whose elements have the same type as the list it receives. Its type annotation must reflect that, by having the same type variable for both the `List` it accepts and the `Array` it returns.

NOTE Elm has three type variable names that have special meanings—number, appendable, and comparable—which we’ll dive into later. For now, avoid choosing any of those as your type variable names!

By the way, you’ll often encounter type variables in documentation with single-letter names like `a`, `b`, `c`, and so on. (For example, the official documentation for `Array.fromList` annotates it as `List a -> Array a`.) You are by no means obliged to do the same! Feel free to use the most self-descriptive names you can think of for your own type variables.

CREATING AN ARRAY OF PHOTOS

Since we want to access our photos by position, we’ll be using `Array.fromList` on our `model.photos` list to translate it into an array.

Let’s add this code below our `initialModel` definition in `PhotoGroove.elm`:

```
photoArray : Array { url : String }
photoArray =
  Array.fromList initialModel.photos
```

If we tried to compile this right now, we’d get an error because we have not imported the `Array` module! Let’s add it to the top of our imports list:

```
import Array exposing (Array)
```

Nice. Let’s verify that our code still compiles by running `elm make src/PhotoGroove.elm --output elm.js` once more before proceeding.

Exposing Imported Types

Notice how we wrote `exposing (Array)` there? If we’d written `import Array` without the `exposing (Array)` part, we’d need to refer to the `Array` type in a qualified style, like so:

```
photoArray : Array.Array { url : String }
```

`Array.Array` refers to “the `Array` type from the `Array` module” just like how `Array.fromList` refers to “the `fromList` function from the `Array` module.” Kinda verbose, right?

By writing `exposing (Array)` after `import Array`, we get to skip the “`Array.`” prefix and use the more concise annotation:

```
photoArray : Array { url : String }
```

Lovely!

3.1.3 Reusing Annotations with Type Aliases

Now that we've created `photoArray`, we can see that our annotations for `initialModel` and `photoArray` have a bit of code duplication going on: they both include `{ url : String }`.

```
initialModel : { photos : List { url : String }, selectedUrl : String }
photoArray : Array { url : String }
```

We can replace this duplication with shared code using a *type alias*.

DEFINITION A *type alias* assigns a name to a type. Anywhere you would refer to that type, you can substitute this name instead.

Let's create a type alias called `Photo`, and then use that in place of `{ url : String }`.

Listing 3.2 Creating a type alias for Photo

```
type alias Photo =      #A
  { url : String }      #A

initialModel : { photos : List Photo, selectedUrl : String }
initialModel =
  ...

photoArray : Array Photo
photoArray =
  Array.fromList initialModel.photos
```

#A “whenever I say `Photo`, I mean `{ url : String }`”

Not only does this make our `initialModel` annotation more concise, it also makes it easier to maintain! Now if we add a new field to our `Photo` record, we can change the type alias in one place instead of having to hunt down several individual annotations.

ANNOTATING INITIALMODEL USING A TYPE ALIAS

Since `initialModel`, `view`, and `update` all involve our model record, let's add a type alias for `Model` as well, and then revise `initialModel`'s type annotation to use it:

```
type alias Model =
  { photos : List Photo
  , selectedUrl : String
  }

initialModel : Model
```

That takes care of `initialModel`. What about `view`?

HTML'S TYPE VARIABLE

We know `view` takes a `Model` and returns `Html`, but we can't just write "`view` returns `Html`" and call it a day. This is because, just like `List` and `Array`, the `Html` type has a type variable!

`Html`'s type variable reflects the type of message it sends to `update` in response to events from handlers like `onClick`.

Table 3.4 Comparing type variables for `List` and `Html`

Value	Type	Description
<code>["foo"]</code>	<code>List String</code>	List of String elements
<code>[3.14]</code>	<code>List Float</code>	List of Float elements
<code>div [onClick "foo"] []</code>	<code>Html String</code>	Html producing String messages
<code>div [onClick 3.14] []</code>	<code>Html Float</code>	Html producing Float messages
<code>div [onClick { x = 3.3 }] []</code>	<code>Html { x : Float }</code>	Html producing { x : Float } messages

Since our `onClick` handler produces messages in the form of records that have an `description` string and a `data` string, our `view` function's return type is:

```
Html { description : String, data : String }
```

That's pretty verbose! Even though it won't remove any code duplication yet, it's perfectly fine to introduce a type alias to make `view`'s annotation more concise. Let's add one above `view`:

```
type alias Msg =
  { description : String, data : String }

view : Model -> Html Msg
```

Excellent! Next we'll look into annotating `view`'s helper function, `viewThumbnail`.

3.1.4 Annotating Longer Functions

We've now seen a few type annotations for functions that take a single argument, but none for functions that take multiple arguments. Our `viewThumbnail` function will be the first.

To learn how to annotate a multi-argument function, let's play around with one that has a simpler return type: the humble `padLeft` function. `padLeft` makes sure strings meet a certain minimum length. You give it a minimum length, a "filler character," and a string. If the string is not at least the given length, `padLeft` adds filler characters to its left until the string reaches that length.

We can try it out in `elm repl`, since `padLeft` is included in Elm's core `String` module:

```
> String.padLeft 9 '.' "not!"
".....not!" : String

> String.padLeft 2 '.' "not!"
"not!" : String
```

We can see that `String.padLeft` takes three arguments—an `Int`, a `Char`, and a `String`—and then returns another `String`. How can we annotate a function like that?

Believe it or not, we’ve already seen the answer! It’s one of those answers that likes to hide in plain sight.

ANNOTATING A PARTIALLY APPLIED FUNCTION

Let’s think back to Chapter 2, where we learned that all functions in Elm are *curried*. That means they all support *partial application*—the practice of calling a function without passing all the arguments it requires. If we call `padLeft` passing only one of its three arguments, the result will be a function that takes the remaining arguments and “finishes the job.”

Here’s a partial application of `padLeft`. What would the annotation for this be?

```
padNine = String.padLeft 9
```

`padLeft 9` is a partial application, so we know `padNine` must be a function. That’s a start!

We also know that `padLeft` takes a `Char` as its next argument after the 9. That gives us enough information to rough out some pseudocode:

```
padNine : Char -> (a function of some sort)
padNine = String.padLeft 9
```

Now let’s suppose we gave this `padNine` function the `Char` it wants. Passing a `'.'` character to `padNine` would be the the same as passing 9 and then `'.'` to the original `padLeft`, like so:

```
padNineDots = String.padLeft 9 '.'
```

How would we annotate `padNineDots`? Since it gives `padLeft` two of three arguments, that means only one more argument is needed to finish the job. That last argument is a `String`, so **`padNineDots` must take a `String` and return a `String`**. We know how to write that one!

```
padNineDots : String -> String
padNineDots = String.padLeft 9 '.'
```

We now know that:

- `padNine` takes a `Char` and returns a function (of some type)
- If you pass `padNine` a `Char`, it returns a `String -> String` function

Putting those two together, our mystery function must be a `String -> String`!

That tells us we can annotate `padNine` like so:

```
padNine : Char -> (String -> String)
padNine = String.padLeft 9
```

Now let's unwind our original partial application. If we cease partially applying the 9, we can follow the pattern we saw here to arrive at a valid annotation for the original `padLeft` itself:

```
padLeft : Int -> (Char -> (String -> String))
```

This is a perfectly accurate annotation, but it's a bit heavy on the parentheses, yeah? Fortunately, we can be more concise. Elm's syntax lets us omit the parentheses here, so the following two annotations are equivalent:

```
padNine : Int -> (Char -> (String -> String))
padNine : Int -> Char -> String -> String
```

Either style works just as well as the other, but the latter is considered best practice. You can see `elm repl` use this annotation style in Listing 3.3, as we partially apply `String.padLeft` step by step, until it has no more arguments to accept and finally returns a `String`.

Listing 3.3 Multi-Argument function annotations

```
> String.padLeft
<function:padLeft> : Int -> Char -> String -> String

> String.padLeft 9
<function> : Char -> String -> String

> String.padLeft 9 '.'
<function> : String -> String

> String.padLeft 9 '.' "not!"
".....not!" : String
```

Notice how each time we partially applied `padLeft`, our annotation got shorter:

Function	Type Annotation
<code>String.padLeft</code>	<code>Int -> Char -> String -> String</code>
<code>String.padLeft 9</code>	<code>Char -> String -> String</code>
<code>String.padLeft 9 '.'</code>	<code>String -> String</code>
<code>String.padLeft 9 '.' "not!"</code>	<code>String</code>

If you think about it, this means technically **every Elm function takes only one argument**.

After all, any function that *appears* to take multiple arguments is ultimately calling out to single-argument functions behind the scenes. The fact that Elm lets you omit the parentheses for these nested calls is just a syntactic convenience.

TIP You can now tell your friends that in Chapter 2 of this book, you wrote an entire working Elm application where every function took only a single argument. This sounds like some really hardcore programming unless your friends are familiar with currying!

ANNOTATING VIEWTHUMBNAIL

Armed with this knowledge, we can follow this pattern to annotate `viewThumbnail` like so:

```
viewThumbnail : String -> Photo -> Html Msg
viewThumbnail selectedUrl thumbnail =
```

Splendid! Our code is getting easier and easier for teammates to pick up at a glance.

3.2 Case-expressions and Custom types

Now that our documentation situation is looking better, let's shift gears to work on those two new features: the *Surprise Me!* button and the thumbnail size chooser.

We'll introduce these iteratively, first adding the visual elements with only a token level of interactivity, and then circling back to make them work properly afterwards.

3.2.1 Using case-expressions

Let's start by adding the *Surprise Me!* button to our `view`, right above the thumbnails `div`.

```
, button
  [ onClick { description = "ClickedSurpriseMe", data = "" } ]
  [ text "Surprise Me!" ]
, div [ id "thumbnails" ]
```

Next let's revise our `update` function to add a quick `else if` branch for our new "ClickedSurpriseMe" message, and a shiny new type annotation while we're at it.

```
update : Msg -> Model -> Model
update msg model =
  if msg.description == "ClickedPhoto" then
    { model | selectedUrl = msg.data }

  else if msg.description == "ClickedSurpriseMe" then
    { model | selectedUrl = "2.jpeg" }

  else
    model
```

This implementation always selects the second photo. That's not much of a *surprise*, granted, but we'll get there. For now, let's smooth out that code duplication we just introduced!

REFACTORED AN IF-EXPRESSION INTO A CASE-EXPRESSION

We now have two conditionals that do nothing more than compare `msg.description` to a string. Let's express this more cleanly by rewriting these *if-expressions* as a *case-expression*:

<pre> case msg.description of "ClickedPhoto" -> { model selectedUrl = msg.data } "ClickedSurpriseMe" -> model selectedUrl = "2.jpeg" } _ -> del </pre>	<div style="border: 1px solid #ccc; padding: 10px; background-color: #f9f9f9;"> <p style="text-align: center; margin: 0;">equivalent <i>if-expression</i></p> <pre> if msg.description == "ClickedPhoto" then { model selectedUrl = msg.data } else if msg.description == "..." then { model selectedUrl = "2.jpeg" } else model </pre> </div>
---	---

Figure 3.2 Refactoring an if-expression into a case-expression

Whereas an *if-expression* is a two-way conditional, a *case-expression* is a multi-way conditional. It lets us compare something to a wider range of values than just `True` or `False`.

Just like with a JavaScript *switch-statement*, we begin a *case-expression* by providing a value which will be run through a series of comparisons. Here we wrote `case msg.description of` because we want to run `msg.description` through these comparisons.

Following the `case` are a series of *branches* such as the following:

```

"ClickedPhoto" ->
  { model | selectedUrl = msg.data }

```

This says that if `msg.description` is equal to `"ClickedPhoto"`, then the branch after the `->` will be evaluated, and the entire *case-expression* will evaluate to that branch's result.

If `msg.description` is not equal to `"ClickedPhoto"`, then the next branch (in this case the `"ClickedSurpriseMe" ->` branch) will be checked in the same way, and so on. Elm has no equivalent of JavaScript's `break` statement, because *case-expression* branches don't "fall through" like branches in JavaScript's *switch-statements* do.

Just like *if-expressions*, *case-expressions* must always evaluate to a single value, meaning exactly one branch must always be chosen for evaluation. When writing *if-expressions* we have `else` to ensure we end up with a value no matter what, whereas in *case-expressions* we can use the default branch of `_ ->` for the same purpose.

NOTE In a *case-expression*, every branch must be indented the same amount—similar to what we saw in Chapter 1 with *let-expressions*. Indentation level is only significant with *let-expressions* and *case-expressions*.

Our code is really shaping up! Let's copy the *case-expression* from Figure 3.2 into `PhotoGroove.elm`, then run `elm make` once more to recompile it.

TRYING OUT THE 'SURPRISE ME!' BUTTON

Before we open up `index.html` and view our progress, we have one last change to make. While we've been hard at work on code quality and new functionality, one of our new teammates has been busy improving our stylesheets.

Let's edit `index.html` to make use of their work, by changing its `<head>` to the following:

```
<head>
  <link rel="stylesheet" href="http://elm-in-action.com/styles.css">
</head>
```

Lovely! Now we can open our revised `index.html` to see the new button in all its glory.

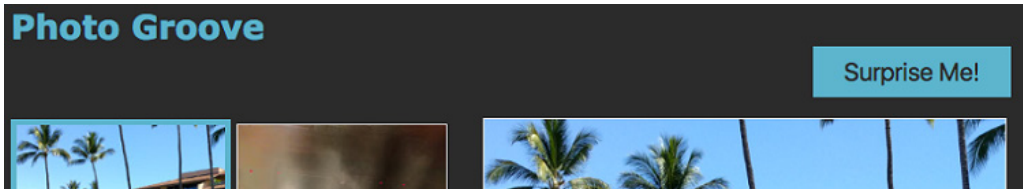


Figure 3.3 The new Surprise Me! button

In addition to styling the *Surprise Me!* button, our helpful teammate went ahead and added styles for the Thumbnail Size Chooser too.

How about we introduce some logic to go with those styles?

3.2.2 Enumerating possibilities with Custom types

Our second feature lets users choose one of three thumbnail sizes: small, medium, or large.

The first step toward implementing it will be storing the current `chosenSize` in our model. In JavaScript we might represent `chosenSize` as a `String`—perhaps setting it to either `"SMALL"`, `"MEDIUM"`, or `"LARGE"`—but in Elm we can do better with *custom types*.

DEFINITION A *custom type* is one you define by specifying the values it can contain.

One use for a custom type is as an enumeration of values. Figure 3.4 illustrates how we can define a custom type to represent the different thumbnail size choices we'll support:

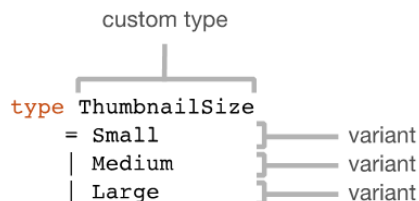


Figure 3.4 Defining an enumeration of values with a custom type called `ThumbnailSize`

This is saying “define a new type called `ThumbnailSize` with three possible values: `Small`, `Medium`, and `Large`.” Let’s add this code to `PhotoGroove.elm`, right above type alias `Msg`.

Let's add the definition of `ThumbnailSize` in Figure 3.4 to our code, right above `type alias Photo`. We can now define `chosenSize` in one of the following ways:

```
chosenSize : ThumbnailSize
chosenSize = Small

chosenSize : ThumbnailSize
chosenSize = Medium

chosenSize : ThumbnailSize
chosenSize = Large
```

Notice that in each of these examples, the type of `chosenSize` is `ThumbnailSize`. These new values we've created—`Small`, `Medium`, and `Large`—aren't "actually integers under the hood" or "just strings behind the scenes." We really have scratch-built a brand new type here! Trying to compare a `ThumbnailSize` to a number, string, or any other type (using `==` or any other comparison) will yield an error at build time.

This is different from `type alias`, which gives a **name** to an existing type—much like how a variable gives a name to an existing value.

These custom type values are also unique. The expression `Medium == Medium` is `True`, but put absolutely any other value (besides `Medium`) on either side of that `==` and the expression will no longer be `True`. The same can be said of `Small == Small` and `Large == Large`.

NOTE Boolean values in Elm are capitalized because `Bool` is a custom type. Its definition looks like this:

```
type Bool = True | False
```

Let's add the `chosenSize` field to our model and initialize it to `Medium` in `initialModel`:

```
type alias Model =
  { photos : List Photo
  , selectedUrl : String
  , chosenSize : ThumbnailSize
  }

initialModel : Model
initialModel =
  { photos = ...
  , selectedUrl = "1.jpeg"
  , chosenSize = Medium
  }
```

Great! Next we'll render some radio buttons to let users change it.

RENDERING A THUMBNAIL SIZE RADIO BUTTON

We can use our newfound knowledge of custom types and *case-expressions* to write a helper function that takes a `ThumbnailSize` and renders a radio button which chooses that size. Let's add the following to `PhotoGroove.elm`, right below `viewThumbnail`:

Listing 3.4 Rendering a thumbnail size radio button

```
viewSizeChooser : ThumbnailSize -> Html Msg
viewSizeChooser size =
  label []
    [ input [ type_ "radio", name "size" ] []
      , text (sizeToString size)
    ]

sizeToString : ThumbnailSize -> String
sizeToString size =
  case size of
    Small -> #A
      "small" #A

    Medium -> #B
      "med" #B

    Large -> #C
      "large" #C
```

#A evaluates to "small" if size == Small
 #B evaluates to "med" if size == Medium
 #C evaluates to "large" if size == Large

NOTE The underscore in that `type_` attribute is very important! As we've seen, `type` is a reserved keyword in Elm (used to define custom types), which means it can't be used for an `Html` attribute name. The `Html.Attributes` module names the attribute `type_` to work around this.

Notice anything missing from that *case-expression*? It has no default branch! We accounted for the cases where `size` is `Small`, `Medium`, and `Large`...but what if `size` has some other value, such as `null`, `undefined`, or `"halibut"`?

Can't happen! `sizeToString`'s type annotation tells us that `size` is guaranteed to be a `ThumbnailSize`, and a `ThumbnailSize` can only be `Small`, `Medium`, or `Large`. It can't be `null` or `undefined` because those don't exist in Elm, and it can't be `"halibut"` because that's a `String`, not a `ThumbnailSize`.

Elm's compiler knows we've covered every possibility here, so it doesn't require a default branch. In fact, if we tried to add one, the compiler would politely inform us that we'd written unreachable code.

RENDERING THREE RADIO BUTTONS

We can now call this `viewSizeChooser` function three times, passing `Small`, `Medium`, and `Large`, to render the three radio buttons on the page. Let's do that right above `thumbnails`:

```
, h3 [] [ text "Thumbnail Size:" ]
, div [ id "choose-size" ]
  [ viewSizeChooser Small, viewSizeChooser Medium, viewSizeChooser Large ]
, div [ id "thumbnails" ]
  (List.map (viewThumbnail model.selectedUrl) model.photos)
```

Once again we see some code duplication: we're calling the same function (`viewSizeChooser`) on every element in a list. Any time we're calling the same function on every element in a list, there's a good chance we can make our code cleaner with `List.map`!

This is absolutely one of those times. Let's refactor our `choose-size` element to this:

```
, div [ id "choose-size" ]
      (List.map viewSizeChooser [ Small, Medium, Large ])
```

Not only does this refactor reduce duplication, it also makes our code more concise.

RENDERING DIFFERENT THUMBNAIL SIZES

Rendering the interface for *choosing* a thumbnail size is great, but it's only half the battle! We still need to actually render our thumbnails differently based on what the user has chosen.

Thanks to our coworker having written a stylesheet for us, we can implement this by adding the class "small", "med", or "large" to our `thumbnails` container. Since those classes conveniently correspond to the results of our `sizeToString` function, we can go ahead and replace our current `div [id "thumbnails"]` with the following:

```
div [ id "thumbnails", class (sizeToString model.chosenSize) ]
```

TIP It would be best practice to write a separate `sizeToClass` function and use it here in place of `sizeToString`, even if their implementations were identical for the time being. Having separate functions means if someone later changes the text on the radio buttons, our thumbnail classes won't accidentally break.

Let's see how it looks! Since we haven't implemented the radio button logic yet, try changing the `chosenSize` value in `initialModel` to `Large` and then recompile to see how that affects the way the page looks. Then try recompiling again with `Small`, and finally back to `Medium`.

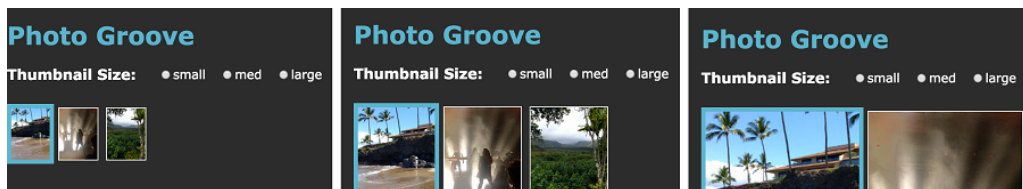


Figure 3.5 Recompiling with `chosenSize = Small`, `chosenSize = Medium`, and `chosenSize = Large`

Cool! Now that we have our interface rendering, it's time to start making the logic work.

3.2.3 Holding Data in Custom Types

Making the *Surprise Me!* button change our model's `selectedUrl` field to a random photo's `url` means we'll need to access a random photo from our `photoArray`.

In JavaScript we might implement reading a photo from an array as follows:

```
var photos = [ ...maybe there are photos here, but maybe not... ];
var selectedId = photos[2].url; // Select the third photo
```

What happens if `photos` doesn't have at least three elements? In that case, `photos[2]` would evaluate to `undefined`, and `photos[2].url` would throw an angry runtime exception such as:

```
Uncaught TypeError: Cannot read property 'url' of undefined
```

Elm avoids this runtime exception by handling this situation differently in three ways:

1. In Elm you call `Array.get` to get an element in an array; there is no `[2]` accessor.
2. `Array.get` never returns `undefined` or `null`, because Elm has neither of these.
3. `Array.get` **always** returns a container value called a `Maybe`.

What's a `Maybe`? Glad you asked!

DEFINITION A `Maybe` is a container like a `List`, except it can hold one element at most.

`Maybe` is implemented as a custom type, but a flavor of custom type we haven't seen yet: one that holds data. Its definition looks like this:

```
type Maybe value
  = Just value
  | Nothing
```

MAYBE VERSUS UNDEFINED

The most common use for `Maybe` is to represent the potential absence of a value. It provides a **container-based** alternative to JavaScript's **drop-in replacements** of `null` and `undefined`.

Figure 3.6 compares Elm's `Array.get` and `Maybe` to JavaScript's `[2]` and `undefined`.

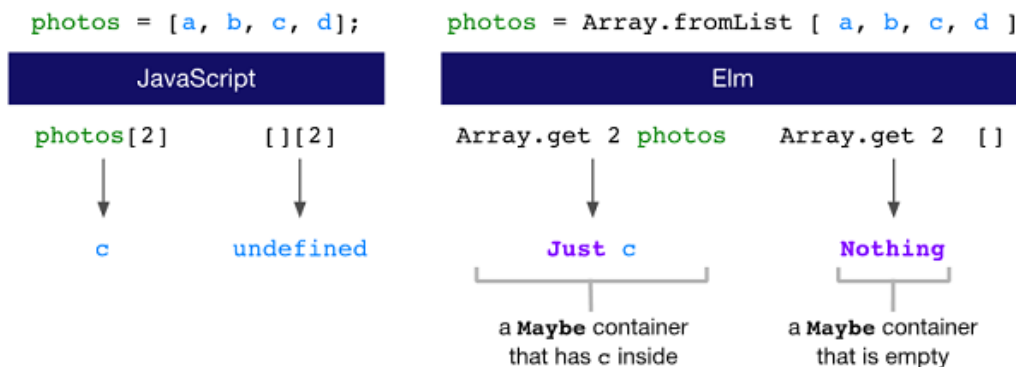


Figure 3.6 Comparing Elm's `Maybe` to JavaScript's `undefined`

`Array.get 2 photos` expresses the desire to access the third element of an array called `photos`. However, the `photos` array may have fewer than three elements in it! If the index 2 is outside the bounds of `photos`, then `Array.get 2 photos` will return `Nothing`.

On the other hand, if 2 is inside its bounds, then `Array.get` will return `Just c`—where `c` is the element at index 2 in the `photos` array.

MAYBE'S TYPE VARIABLE

As we've seen before with `List`, `Array`, and `Html`, the `Maybe` type has a type variable!

We chose to name it `value` in the definition we gave for `Maybe`'s custom type earlier:

```
type Maybe value
  = Just value
  | Nothing
```

Similarly to how you can have a `List String` or a `List Photo`, you can also have a `Maybe String` or `Maybe Photo`. The difference is that whereas `List Photo` means "a list of photos," `Maybe Photo` means "either a `Photo` or nothing at all." Put another way, `Maybe` is a container that can hold at most one element.

Now that we know `Just "1.jpeg"` is a `Maybe String`, what is the type of `Just` itself?

"JUST" IS A FUNCTION

Argh, that heading totally gave away the answer. Oh well.

As the heading suggests, whereas `Nothing` is a `Maybe` value, **`Just` is a function** that returns a `Maybe` value. We can confirm this in `elm repl`:

```
> Nothing
Nothing : Maybe.Maybe a

> Just
<function> : a -> Maybe.Maybe a

> Just "dance"
Just ("dance") : Maybe.Maybe String
```

NOTE `elm repl` calls the type `Maybe.Maybe` because the `Maybe` type lives in a module called `Maybe`. This is similar to `Array.Array` from earlier, except that unlike the `Array` module, the `Maybe` module is imported by default. (The `Maybe.Maybe` type is not exposed by default.)

In Chapter 1 we noted that when you write a function by hand, its name must be lowercase...yet here we see that custom type variants (like `Just`) are functions with capitalized names. What makes them so special?

A capital question! The answer is right around the corner.

DESTRUCTURING CUSTOM TYPES

In Chapter 1 we learned how we can *destructure* tuples to both extract and name their values.


```
multiply3d ( x, y, z ) = x * y * z
```

As it turns out, we can also destructure custom type variants such as `Just` in the branches of *case-expressions*. This destructuring is what sets variants apart from other functions! Let's use what we've learned about `Array.get` and `Maybe` to add this `getPhotoUrl` function right above our `update` function.

Listing 3.5 Selecting a photo by index

```
getPhotoUrl : Int -> String
getPhotoUrl index =
  case Array.get index photoArray of
    Just photo -> #A
                  photo.url
    Nothing ->
      "" #B
```

#A Destructuring `Just`` and naming its contained value `photo``
 #B Fall back on `""` if there was no photo at that index

Here, `Just photo ->` is saying two things:

1. This branch matches a `Maybe` value that was created using the `Just` type variant.
2. We're extracting the value that was passed to `Just` and naming it `photo`.

NOTE This is where the distinction between capitalized and uncapitalized functions matters. By comparing their capitalizations, Elm's compiler can tell that `Just photo ->` refers to a type variant called `Just` which holds a value we've chosen to name `photo`. If we'd instead written `Just True ->` the compiler would know we meant "the `Just` type variant holding exactly the value `True`."

Notice that this *case-expression* did not need a `_ ->` branch! This is because its branches already cover all possible values of `Maybe`. One branch covers `Nothing`, and the other covers any values created with `Just`. Since `Nothing` and `Just` are the only ways to obtain a `Maybe`, these branches have every possibility covered.

Also notice how because `Array.get` returns a `Maybe`, you can't help but remember to handle the case where `index` is outside the bounds of `photoArray`. This API design means that you'll always be doing the minimal amount of coding necessary to handle error cases. You don't need to defensively check whether something is `null` or `undefined`, because Elm's compiler has your back in cases where the value you want might not be available.

3.2.4 Representing Flexible Messages with Custom types

We've made so much progress! We've updated our views to render the new interfaces, expanded our model to hold our new `chosenSize` field, and introduced helper functions for `photoArray`. Now it's time to wire all that together with changes to `update`.

Let's start by updating that `chosenSize` field, which currently gets initialized to `Medium` and then never changes. We'll want to update it whenever the user clicks a radio button.

This might seem like the obvious change to make to `viewSizeChooser`:

```
viewSizeChooser : ThumbnailSize -> Html Msg
viewSizeChooser size =
  ...
  onClick { description = "ClickedSize", data = size }
```

There's just one problem...this is a type mismatch! Remember our `type alias` for `Msg`?

```
type alias Msg =
  { description : String, data : String }
```

This says the `data` field of a `Msg` needs to be a `String`, not a `ThumbnailSize`! To preserve backwards compatibility with existing code, we need a `Msg` type that can accommodate both `ThumbnailSize` and `String`. One way we can do this is by adding a new field called `size`:

```
type alias Msg =
  { description : String, data : String, size : ThumbnailSize }
```

This solves one problem while creating another. Now our existing `onClick` handler will no longer compile, because it's missing a field. Here's how we wrote it back in Chapter 2:

```
onClick { description = "ClickedPhoto", data = thumbnail.url }
```

Without specifying a value for that new `size` field, this record is no longer a valid `Msg`. We'd need to change it to something like this:

```
onClick { description = "ClickedPhoto", data = thumbnail.url, size = Small }
```

Yuck. Having to set `size = Small` for a message that doesn't care about `size`? That doesn't smell right. And are we really planning to add a field to every `Msg` in our program whenever we need it to support a new data type? This doesn't seem like it will scale well.

Is there a better way?

IMPLEMENTING MSG AS A CUSTOM TYPE

There totally is! Implementing `Msg` as a custom type will work much better. Here's the plan:

1. Replace our `type alias Msg` declaration with a `type Msg` declaration.
2. Revise `update` to use a *case-expression* which deconstructs our new custom type.
3. Change our `onClick` handler to pass a type variant instead of a record.

Listing 3.6 shows the first two changes: replacing `type alias Msg` with `type Msg`, and revising `update` accordingly. Let's edit `PhotoGroove.elm` to incorporate these changes.

Listing 3.6 Implementing Msg as a custom type

```
type Msg
  = ClickedPhoto String
  #A
  #B
```

```

    | ClickedSize ThumbnailSize #C
    | ClickedSurpriseMe         #D

update : Msg -> Model -> Model
update msg model =
  case msg of
    ClickedPhoto url -> #E
      { model | selectedUrl = url }

    ClickedSurpriseMe -> #F
      { model | selectedUrl = "2.jpeg" }

```

#A Replaces our earlier declaration of type alias Msg
 #B Replaces our "ClickedPhoto" message
 #C Our new message: user clicked a thumbnail size
 #D Replaces our "ClickedSurpriseMe" message
 #E Previous condition: `msg.description == "ClickedPhoto"`
 #F Nothing to destructure for SurpriseMe!

This change means our `onClick` handlers now expect type variants instead of records. We'll need to make this change in `view`:

```

Old: onClick { description = "ClickedSurpriseMe", data = "" }
New: onClick ClickedSurpriseMe

```

Let's also make this change in `viewThumbnail`:

```

Old: onClick { description = "ClickedPhoto", data = thumbnail.url }
New: onClick (ClickedPhoto thumbnail.url)

```

THE MISSING PATTERNS ERROR

If we recompile, we'll see a type of error we haven't seen before: a *missing patterns* error.

```

-- MISSING PATTERNS ----- src/PhotoGroove.elm

This `case` does not have branches for all possibilities.
...
Missing possibilities include:

    ClickedSize _

```

I would have to crash if I saw one of those. Add branches for them!

Oops! The compiler noticed we only handled `ClickedPhoto` and `ClickedSurpriseMe` in our `update` function's *case-expression*; we never wrote the logic for `ClickedSize`. Fortunately, that problem will never reach our end users because the compiler *didn't let us forget*.

TIP Anytime you use the default case `_ ->` in a *case-expression*, you cannot get this error. That is not a good thing! The *missing patterns* error is your friend. When you see it, it's often saving you from a bug you would have had to hunt down later. Try to use `_ ->` only as a last resort, so you can benefit from as many *missing pattern* safeguards as possible.

Let's add this to our `update` function, right above the `ClickedSurpriseMe` branch:

```
ClickedSize size ->
  { model | chosenSize = size }

ClickedSurpriseMe index ->
  { model | selectedUrl = "2.jpeg" }
```

Finally, we need to add a new `onClick` handler to our radio button in `viewSizeChooser`:

```
Old: input [ type_ "radio", name "size" ] []
New: input [ type_ "radio", name "size", onClick (ClickedSize size) ] []
```

If we recompile and click the radio buttons, we can now see the thumbnail sizes change!

TIP There are at least two ways we could improve user experience here. One way would be making the “medium” option display as selected on page load. Another would be to use a broader event handler than `onClick`—one that detects whenever the radio state changes, even if it not from a click. Try implementing these improvements sometime for practice!

Using custom types for `Msg` has two major advantages over records.

1. The compiler can save us from typos. Before, if we wrote `"ClickedPhoti"` instead of `"ClickedPhoto"`, our code would silently fail and we'd have to hunt down the cause. Now, if we write `ClickedPhoti` instead of `ClickedPhoto`, Elm's compiler will give us an error at build time—including the line number of the typo. No bug hunting necessary!
2. Each flavor of `Msg` now holds only the minimum amount of data it needs. `ClickedPhoto` holds only a `String`, and `ClickedSize` holds only a `ThumbnailSize`. Best of all, `ClickedSurpriseMe` doesn't need to hold any data at all...and so it doesn't!

To sum up, implementing `Msg` as a custom type has made our code more reliable, more concise, and easier to scale. These advantages make custom types the typical choice for representing messages in production Elm applications, and they're what we will use for the rest of the book.

Now all that remains is to introduce random number generation into *Surprise Me!*

3.3 Generating Random Numbers with Commands

Currently our *Surprise Me!* button always selects the second photo, but we want it to select a random photo instead. Here's how we're going to do that:

1. Generate a random integer between 0 and 2. This will be our index.
2. Ask `photoArray` for the photo it's storing at that index.
3. Set our model's `selectedUrl` to be that photo's `url`.

We'll start by generating the random integer.

3.3.1 Describing Random Values with `Random.Generator`

In JavaScript, `Math.random()` is the typical starting point for generating random values. Calling `Math.random()` gives you a random `Float` between 0 and 1. To randomly generate an `Int` or a `String`, you manually convert that `Float` into the value you actually want.

In Elm we start with a `Random.Generator`, which specifies the type of value we want to randomly generate. For example, the `Random.int` function takes a lower bound and an upper bound, and returns a `Random.Generator` that generates a random `Int` between those bounds.

Here's a `Generator` that randomly generates integers between 0 and 2:

```
randomPhotoPicker : Random.Generator Int
randomPhotoPicker =
  Random.int 0 2
```

Notice how the type annotation says `Random.Generator Int`. This means "we have a `Random.Generator` that produces `Int` values." If we'd used `Random.bool` instead of `Random.int`, it would have returned a `Random.Generator Bool` instead.

GENERATING A RANDOM PHOTO INDEX

Since `randomPhotoPicker` generates random integers between 0 and 2, and `photoArray` has 3 photos in it, we could use this generator to pick an index within `photoArray` to select. That would work, but it would be brittle since it relies on `photoArray` having exactly 3 photos. We can improve it by replacing the hardcoded 2 with `(Array.length photoArray - 1)`.

Let's add the following after our `getPhotoUrl` declaration.

```
randomPhotoPicker : Random.Generator Int
randomPhotoPicker =
  Random.int 0 (Array.length photoArray - 1)
```

INSTALLING PACKAGE DEPENDENCIES

We also need to add the `Random` module to the end of our imports.

```
import Random
```

This won't quite compile yet, because although we've added `import Random`, the `Random` module is not among the standard modules that were installed when we ran `elm init` in Chapter 2. For `import Random` to work, we first need to install the *package* that contains the `Random` module.

DEFINITION An Elm *package* is an installable collection of modules.

The `elm install` command downloads and installs packages when you give it the name of the package you want. Package names consist of a username followed by a `/` and then the package name; in this case, the package we seek is named `elm/random`.

Let's use `elm install` to get some `Random` going! Run this command in the terminal:

```
elm install elm/random
```

You should see something like this:

Here is my plan:

```
Add:
  elm/random  1.0.0
```

Would you like me to update your elm.json accordingly? [Y/n]:

Answer `y`, and you should shortly see the text `Dependencies ready!` (We'll dig into the `elm.json` file it mentioned in future chapters.)

Now that we've described the random value we want using a `Random.Generator`, added `import Random` to access the module where `Random.Generator` lives, and used `elm install` to obtain the `elm/random` package which houses that module, our code will compile again. We're ready to generate some random values!

3.3.2 Introducing Commands to the Elm Architecture

If you call JavaScript's `Math.random()` five times, you'll probably get back five different numbers. Elm functions are more consistent. If you call any Elm function five times with the same arguments, you can expect to get the same return value each time. This is no mere guideline, but a language-level guarantee! Knowing that all Elm functions have this useful property makes bugs easier to track down and reproduce.

Since Elm forbids functions from returning different values when they receive the same arguments, it's not possible to write an inconsistent function like `Math.random` as a plain Elm function. Instead, Elm implements random number generation using a *command*.

DEFINITION A *command* is a value that describes an operation for the Elm Runtime to perform. Unlike calling a function, running the same command multiple times can have different results.

When the user clicks *Surprise Me!* we'll use a command to translate our `Random.Generator Int` into a randomly-generated `Int`, which will represent our new selected photo index.

RETURNING COMMANDS FROM UPDATE

Remember how we specified what `onClick` should do in terms of a message that got sent to our `update` function? We didn't say "add this click event listener to the DOM right away," we said "I want this `Msg` to get sent to my `update` function whenever the user clicks here."

Commands work similarly. We don't say "generate a random number right this instant," we say "I want a random number, so please generate one and send it to my `update` function gift-wrapped in a `Msg`." As with `onClick`, we let `update` take it from there—and if we so desire, `update` can return new commands which trigger new calls to `update` when they complete.

Figure 3.7 shows how commands flow through the Elm Architecture.

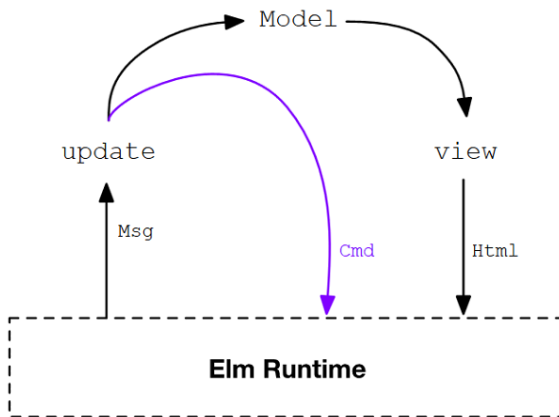


Figure 3.7 How commands flow through the Elm Architecture

Importantly, the addition of commands has not altered the fundamental structure of the Elm Architecture we learned in Chapter 2. All of the following still hold true.

- Our `Model` value is still the single source of truth for the application state.
- Our `update` function is still the only way to alter that model.
- Sending a `Msg` to `update` remains the only way to tell `update` what we want done.

All we've done is to give `update` some new powers: the ability to run logic that can have inconsistent results, such as generating a random number. As we will soon see, the key to unlocking these powers is upgrading from `Browser.sandbox` to `Browser.element`.

RUNNING COMMANDS IN RESPONSE TO USER INPUT

We want to generate our random number in response to a click event, so we'll need to alter our `update` function a bit. Specifically we'll have `update` return a tuple containing not only the new `Model` we want, but also whatever commands we want the Elm Runtime to execute.

Before we edit `update`'s implementation, let's revise its type annotation to guide our work.

```
update : Msg -> Model -> ( Model, Cmd Msg )
```

As you can see, tuple annotations look just like tuple values. `(Model, Cmd Msg)` means "a tuple where the first element is a `Model` and the second is a `Cmd Msg`."

NOTE We write `Cmd Msg` instead of just `Cmd` for the same reason that we write `Html Msg` instead of `Html`. Like `Html`, the `Cmd` type also has a type variable, and for a familiar reason: whenever a `Cmd` finishes, a `Msg` value is what it sends to `update` for further processing.

Let's revise our `update` implementation to reflect this new reality.

```

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    ClickedPhoto url ->
      ( { model | selectedUrl = url }, Cmd.none )

    ClickedSize size ->
      ( { model | chosenSize = size }, Cmd.none )

    ClickedSurpriseMe ->
      ( { model | selectedUrl = "2.jpeg" }, Cmd.none )

```

A fine start! Next we'll replace that `Cmd.none` in `ClickedSurpriseMe`'s branch with a command to randomly generate the index of the photo we want to select.

3.3.3 Generating Random Values with `Random.generate`

The `Random.generate` function returns a command that generates random values wrapped up in messages. It's just what the doctor ordered here! `Random.generate` takes two arguments:

1. A `Random.Generator`, which specifies the type of random value we want.
2. A function which can wrap the resulting random value in one of our `Msg` values.

The `randomPhotoPicker` we created earlier will do nicely for the `Random.Generator`, so all we're missing is a way to wrap these randomly-generated integers in a `Msg`. We can introduce one by adding a new variant to `Msg` and a new branch to `update`:

```

type Msg
  = ClickedPhoto String
  | GotSelectedIndex Int
  ...

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    GotSelectedIndex index ->
      ( { model | selectedUrl = getPhotoUrl index }, Cmd.none )
    ...

```

Great! Next we'll get `Random.generate` to produce one of these `GotSelectedIndex` messages.

CALLING `RANDOM.GENERATE`

When a user clicks the *Surprise Me!* button, the `ClickedSurpriseMe ->` branch of our `update` function's *case-expression* gets run. Currently that returns a `Cmd.none`, but now we want it to return a different command: one that generates a random `Int` and sends it back to `update` wrapped in a `GotSelectedIndex`. We'll obtain this command by calling `Random.generate`.

Earlier we noted that `Random.generate` needs two ingredients:

1. A `Random.Generator`, which specifies what type of random value we want. (We'll use `randomPhotoPicker`, our `Random.Generator Int`, for this.)
2. A function which can wrap the resulting random value in one of our `Msg` values. (We'll use our `GotSelectedIndex` variant for this, since it is a function that returns a `Msg`.)

Let's revise our `ClickedSurpriseMe` branch to return an unchanged model, and to call `Random.generate` instead of returning `Cmd.none`.

```
ClickedSurpriseMe ->
  ( model, Random.generate GotSelectedIndex randomPhotoPicker )
```

Figure 3.8 illustrates how this would cause data to flow through our application, assuming the Elm Runtime randomly generated a `2` after the user clicked our *Surprise Me!* button.

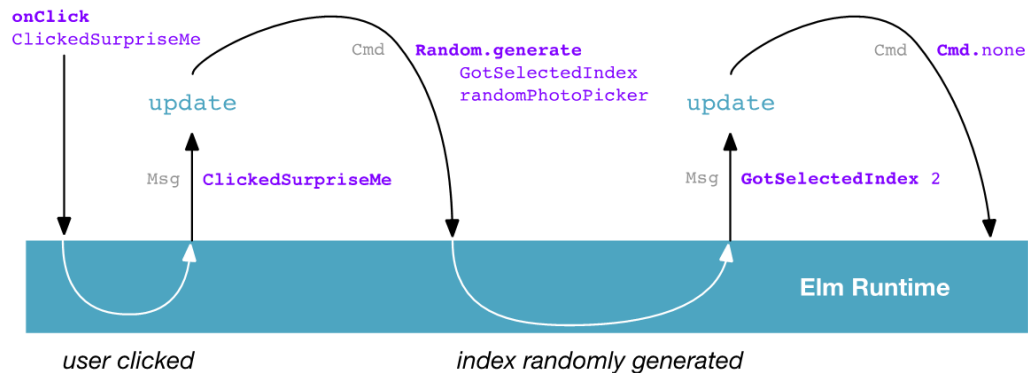


Figure 3.8 Data flowing through our application, assuming the Elm Runtime randomly generated a `2`

TIP Since `update` returns a tuple of both a new `Model` as well as a command, it's possible for the same `update` branch to both change the `Model` and to run a command as well! In such a case, first `Model` would change, then `view` would get run on the new `Model` to update the page, and finally the command would run.

Calling `Random.generate GotSelectedIndex randomPhotoPicker` returns a command which:

1. Randomly generates an `Int` between 0 and 2, because that's the type of random value `randomPhotoPicker` specified it should generate.
2. Takes that randomly-generated `Int`, passes it to `GotSelectedIndex` (an `Int -> Msg` function), and runs the resulting `Msg` through `update`.

We could write a type annotation for `Random.generate` like so:

```
generate : ( randomValue -> msg ) -> Random.Generator randomValue -> Cmd msg
```

NOTE Capitalization is important! There is a big difference between `Msg` and `msg`. A function that returns `Msg` returns an instance of the exact `Msg` custom type we defined in `PhotoGroove.elm`. In contrast, `msg` is a type variable. A function that returns `msg` could return anything at all!

Table 3.5 shows how the expressions involved in our call to `Random.generate` relate to the parameters and type variables in this annotation.

Table 3.5 Types involved in calling `Random.generate`

Expression	Type	Type in Original Annotation
<code>GotSelectedIndex</code>	<code>(Int -> Msg)</code>	<code>(randomValue -> msg)</code>
<code>randomPhotoPicker</code>	<code>Random.Generator Int</code>	<code>Random.Generator randomValue</code>
<code>Random.generate</code> <code>GotSelectedIndex</code> <code>randomPhotoPicker</code>	<code>Cmd Msg</code>	<code>Cmd msg</code>

NOTE Because it takes a function as an argument, `Random.generate` is another *higher-order function* like `List.map` or `String.filter`.

UPGRADING MAIN TO USE `BROWSER.ELEMENT`

We're almost done, but if we compile our code right now, we'll get an error. This is because `Browser.sandbox` wants `update` to return a `Model`, whereas our `update` function now returns a `(Model, Cmd Msg)` tuple. Fortunately `sandbox`'s older sibling, `Browser.element`, expects just such a tuple! Let's replace our `main` declaration with this:

```
main =
  Browser.element
    { init = \flags -> ( initialModel, Cmd.none )
    , view = view
    , update = update
    , subscriptions = \model -> Sub.none
    }
```

We're passing two things differently from what we passed to `Html.beginnerProgram`.

1. The `init` record has been replaced by a function that takes `flags` and returns a tuple. We'll revisit `flags` in Chapter 5; let's focus on the tuple for now. The first element in that tuple is our initial `Model`, and the second is a command to run when the application loads. (`init` is the only place besides `update` that can specify commands to run.) Since we have nothing to run on startup, we write `Cmd.none`.
2. We've added a `subscriptions` field. Like `flags`, we'll also dive into that in Chapter 5; for now let's disregard it.

ANNOTATING MAIN

Finally we'll give `main` the following type annotation:

```
main : Program () Model Msg
```

Whoa! What’s that funky `()` thing?

The `()` value is known as *Unit*. It contains no information whatsoever. It’s both a value and a type; the `()` type can only be satisfied with the `()` value. We could use it to write a function like this:

```
getUltimateAnswer : () -> Int
getUltimateAnswer unit =
  40 + 2
```

The only way to call `getUltimateAnswer` would be to pass `()` to it; since it accepts `()` as an argument, no other value but `()` will do. A function which takes only `()` as an argument returns the same exact value every time it’s called, making it a very boring function.

The reason we see `()` in our `Program` annotation is that `Program`’s three type parameters represent the following pieces of information:

1. Our `flags` type. `flags` refers to the argument `init` receives, which we aren’t currently using. Using `()` for our `flags` type indicates that we don’t accept any flags.
2. Our model’s type. We can have any type of model we like—it could be a `String`, or an `Int`, or any number of other types. In our case, our model has the type `Model`.
3. The type of the message that both `update` and `view` will use. In our case, that type is `Msg`.

Putting these together, we can read `Program () Model Msg` as “an Elm `Program` with no `flags`, whose model type is `Model` and whose message type is `Msg`.”

THE FINAL BUILD

If you recompile with `elm make` and open `index.html`, you should experience a gloriously functional application. Try clicking *Surprise Me!* to watch it randomly pick a different photo!

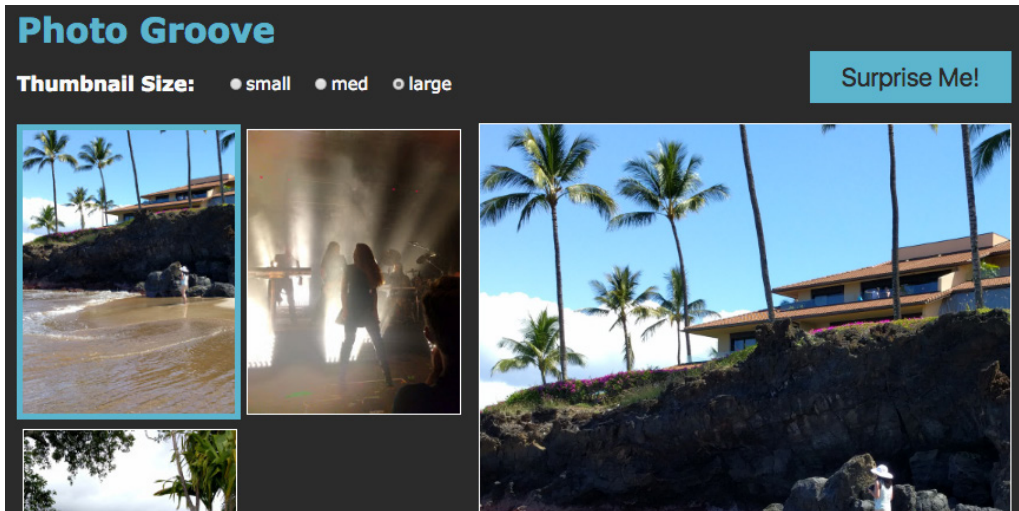


Figure 3.9 Randomly choosing the initially selected photo

Ain't it grand? Even after all the changes we made, once it compiled it just worked—no regressions! Not only that, but our code never came anywhere near causing a runtime exception. This is a normal experience with Elm, yet it never seems to get old.

Compared to where we were at the end of Chapter 2, our revised application is not only more feature-rich, it's also more reliable and better-documented. It does more, it does it better, and the code is cleaner. Huzzah!

3.4 Summary

We've now improved on our application from Chapter 2 in several ways.

- We added documentation in the form of type annotations
- Users can select from one of three thumbnail sizes
- It has a *Surprise Me!* button which selects a thumbnail at random
- From now on whenever we add a new `Msg` value, the compiler will give us a *Missing Patterns* error if we forget to handle it (like when we didn't account for `ClickedSize`.)

Along the way we covered many new concepts, such as:

- Type variables represent concrete types that have not been specified yet.
- A `type alias` declaration assigns a name to a type, much like how a constant assigns a name to a value.
- A `type` declaration defines a new custom type, a custom type that did not exist before.
- Custom types can hold more flexible data than records or tuples can.
- Custom type variants can be either values which are instances of that custom type, or

functions which return instances of that custom type.

- You can destructure custom type variants in *case-expressions* to extract their data.
- If you don't write a fallback `_ ->` branch in a *case-expression*, you'll get a compiler error unless your code handles all possible cases.
- `Array.get` prevents runtime crashes by returning a `Maybe` instead of a normal element.
- The `()` type (known as "Unit") is both a type and a value. The only value of type `()` is the value `()`.
- The type `Program () Model Msg` refers to an Elm `Program` with no flags, whose model type is `Model` and whose message type is `Msg`.

We also learned some differences between comments and type annotations.

Table 3.6 Documenting Code with Comments and Type Annotations

Comment	Type Annotation
Arbitrary string, can describe anything	Can only describe a thing's type
Can be inaccurate or get out of date	Compiler guarantees it's accurate and up-to-date
Can be written just about anywhere	Always goes on the line above the thing it documents

Finally, we learned about `Maybe`, the container value which `Array.get` returns to represent the potential absence of a value. Table 3.7 compares `Maybe` and `List`.

Table 3.7 Comparing Maybe and List

Container Contents	List	Maybe
Empty	<code>[] : List a</code>	<code>Nothing : Maybe a</code>
One value	<code>["foo"] : List String</code>	<code>Just "foo" : Maybe String</code>
Two values	<code>["foo", "bar"] : List String</code>	<i>not possible!</i>

In Chapter 4 we'll take our application *to the extreme*, by talking to a server to obtain our list of photos to display. Stay tuned!

Listing 3.7 The complete PhotoGroove.elm

```
module PhotoGroove exposing (main)

import Array exposing (Array)
import Browser
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onClick)
import Random
```

```

urlPrefix : String                                #A
urlPrefix =
    "http://elm-in-action.com/"

type Msg                                           #B
    = ClickedPhoto String                         #C
    | GotSelectedIndex Int                       #C
    | ClickedSize ThumbnailSize                  #C
    | ClickedSurpriseMe                         #D

view : Model -> Html Msg                          #E
view model =
    div [ class "content" ]
        [ h1 [] [ text "Photo Groove" ]
        , button
            [ onClick ClickedSurpriseMe ]         #F
            [ text "Surprise Me!" ]
        , h3 [] [ text "Thumbnail Size:" ]
        , div [ id "choose-size" ]
            (List.map viewSizeChooser [ Small, Medium, Large ])
        , div [ id "thumbnails", class (sizeToString model.chosenSize) ]
            (List.map (viewThumbnail model.selectedUrl) model.photos)
        , img
            [ class "large"
            , src (urlPrefix ++ "large/" ++ model.selectedUrl)
            ]
            []
        ]

viewThumbnail : String -> Photo -> Html Msg        #G
viewThumbnail selectedUrl thumbnail =
    img
        [ src (urlPrefix ++ thumbnail.url)
        , classList [ ( "selected", selectedUrl == thumbnail.url ) ]
        , onClick (ClickedPhoto thumbnail.url)    #H
        ]
        []

viewSizeChooser : ThumbnailSize -> Html Msg
viewSizeChooser size =
    label []
        [ input [ type_ "radio", name "size", onClick (ClickedSize size) ] []
        , text (sizeToString size)
        ]

sizeToString : ThumbnailSize -> String
sizeToString size =
    case size of
        Small ->                                     #I
            "small"                                   #I
        Medium ->                                     #I

```

```

        "med"                                #I
                                           #I
        Large ->                             #I
        "large"                             #I

type ThumbnailSize
= Small
| Medium
| Large

type alias Photo =                          #J
{ url : String }                           #J

type alias Model =
{ photos : List Photo                      #K
, selectedUrl : String                    #K
, chosenSize : ThumbnailSize              #K
}

initialModel : Model
initialModel =
{ photos =
  [ { url = "1.jpeg" }
  , { url = "2.jpeg" }
  , { url = "3.jpeg" }
  ]
, selectedUrl = "1.jpeg"
, chosenSize = Medium
}

photoArray : Array Photo
photoArray =
  Array.fromList initialModel.photos

getPhotoUrl : Int -> String
getPhotoUrl index =
  case Array.get index photoArray of
    Just photo ->                          #L
      photo.url                             #L
    Nothing ->                             #L
      ""                                    #L

randomPhotoPicker : Random.Generator Int   #M
randomPhotoPicker =                        #M
  Random.int 0 (Array.length photoArray - 1) #M

update : Msg -> Model -> ( Model, Cmd Msg ) #N
update msg model =
  case msg of

```

```

GotSelectedIndex index ->
  ( { model | selectedUrl = getPhotoUrl index }, Cmd.none )

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

ClickedSize size ->
  ( { model | chosenSize = size }, Cmd.none )

ClickedSurpriseMe ->
  ( model, Random.generate GotSelectedIndex randomPhotoPicker ) #O

main : Program () Model Msg                                     #P
main =
  Browser.element
  { init = \flags -> ( initialModel, Cmd.none )
  , view = view
  , update = update
  , subscriptions = \model -> Sub.none
  }

```

#A “The type of urlPrefix is String”

#B Msg is a custom type with 4 variants

#C These variants are containers. They each hold a different type of value.

#D This variant is not a container. It holds no extra information.

#E view is a function that takes a Model and returns a Html Msg value.

#F The ClickedSurpriseMe variant is not a function.

#G Arguments are separated by -> because they can be partially applied.

#H The ClickedPhoto variant is a function. Its type is String -> Msg.

#I This case-expression has no default branch because these three branches already cover all possibilities.

#J A type alias gives a name to a type. Anywhere we see “Photo” we could have written this instead.

#K Type aliases can use other type aliases like Photo, as well as custom types like ThumbnailSize.

#L Array.get returns a Maybe because it’s possible that there is no value at that index.

#M This Random.Generator Int describes a random number generator that produces integers.

#N Tuples are shorthand for records. (Model, Cmd Msg) is a tuple with two elements.

#O Random.generate returns a Cmd that generates a random value, wraps it in GotSelectedIndex, and passes it to update.

#P An Elm program with no flags, with Model for its model type and Msg for its message type.

4

Talking to Servers

This chapter covers

- Using Decoders to validate and translate JSON
- Handling descriptive errors with Results
- Communicating over HTTP with Requests

We've made great progress! Users of our *Photo Groove* application can now do quite a bit:

- Select a thumbnail to view a larger version
- Choose Small, Medium, or Large thumbnail sizes
- Click *Surprise Me!* to select a random thumbnail

Our manager is impressed with our progress, but has even more in mind. "It's time to take *Photo Groove* to the next level. The highest level. The *cloud* level. That's right, we're going to have *Photo Groove* start getting its photo information from our very own servers!"

As with last time, our manager has one more minor feature request. "There's also going to be some metadata associated with each photo—specifically download size and an optional caption. We can have it show those on top of the big photo, right?"

Sure, since you asked so nicely.

In this chapter we'll teach our application how to talk to servers. We'll validate and translate JSON data, communicate over HTTP using the Elm Architecture, and reliably handle errors in client-server communication.

Let's get to it!

4.1 Preparing for Server-Loaded Data

Since we're going to be loading our initial list of photos from the server, our data modeling needs will change.

4.1.1 Modeling Incremental Initialization

Right now our `initialModel` looks like this:

```
initialModel : Model
initialModel =
  { photos = [ { url = "1.jpeg" }, { url = "2.jpeg" }, { url = "3.jpeg" } ]
  , selectedUrl = "1.jpeg"
  , chosenSize = Medium
  }
```

We'll be loading our photos from the server, meaning `initialModel.photos` will be `[]` and we'll update that list once we hear back from the server. This will have a few consequences:

1. Because we have no initial photos, we can't have an initial `selectedUrl` either.
2. The initial request to load the data might fail, for example because the server is down. In that case, we should display an error.
3. Our `photoArray` value references `initialModel.photos`, so it will be affected.
4. Both `getPhotoUrl` and `randomPhotoPicker` reference `photoArray`, so they will be affected as well.

Let's translate these new considerations into data model changes!

MODELING ABSENT VALUES WITH MAYBE

The `Maybe` container we learned about in Chapter 3 seems like a nice way to implement the first consideration—that we won't have an initial `selectedUrl`. Since `Maybe` represents the potential absence of a value—by holding either the value `Nothing` or some other value wrapped in a `Just`—it's an intuitive way to represent the data that is absent until we hear back from the server.

We can also use a `Maybe` to represent a potential loading error message. It will begin as `Nothing`, but might become something like `Just "Server unavailable"` if we get an error.

Go ahead and change `Model` and `initialModel` to look like the following.

```
type alias Model =
  { photos : List Photo
  , selectedUrl : Maybe String
  , loadingError : Maybe String
  , chosenSize : ThumbnailSize
  }

initialModel : Model
initialModel =
  { photos = []
  , selectedUrl = Nothing
  , loadingError = Nothing
  , chosenSize = Medium
  }
```

Now that we've revised our data model, we can lean on the compiler to tell us what parts of our code base were affected by this revision. This will let us reliably propagate these changes throughout our code base!

PROPAGATING CHANGES

If we recompile we'll see two type mismatches: one in `view` and one in `update`.

Since `selectedUrl` is now a `Maybe String` instead of a `String`, our `SelectByUrl` logic inside `update` now needs a `Just`. Let's add it:

```
SelectByUrl url ->
  ( { model | selectedUrl = Just url }, Cmd.none )
```

Our `SelectByIndex` logic has a similar type mismatch, but there we can actually do better! One way to fix the mismatch is to wrap what `getPhotoUrl` returns in a `Just`...but another way is to propagate the `Maybe` itself! Let's rewrite `getPhotoUrl` to return a `Maybe`:

```
getPhotoUrl : Int -> Maybe String
getPhotoUrl index =
  case Array.get index photoArray of
    Just photo ->
      Just photo.url
    Nothing ->
      Nothing
```

Notice how this not only resolves the type mismatch, but also means we are no longer returning the clearly invalid URL of "" in the case where the index is out of bounds. One less potential bug to worry about!

FIXING VIEWTHUMBNAIL

Now that we've resolved our type mismatches in `update`, we can move on to resolving the ones in `view`. One of the problems in `view` is that the `viewThumbnail` function it calls has an inaccurate type annotation. It says it's taking a `String` called `selectedUrl`, but `selectedUrl` is now a `Maybe String`. Let's fix that annotation real quick:

```
viewThumbnail : Maybe String -> Photo -> Html Msg
viewThumbnail selectedUrl thumbnail =
```

There's another problem in `viewThumbnail`: the code `selectedUrl == thumbnail.url` is now comparing a `String` to a `Maybe String`, which is a type mismatch! The easiest way to fix this is by calling `Just` on `thumbnail.url`, converting it to a `Maybe String`. Now we're comparing a `Maybe String` to a `Maybe String` and balance has been restored to the galaxy.

```
classList [ ( "selected", selectedUrl == Just thumbnail.url ) ]
```

FIXING THE PREVIEW IMAGE

With those “quick fixes” out of the way, we can recompile to see one final error: the code that renders the large preview image has a type mismatch.

```
src (urlPrefix ++ "large/" ++ model.selectedUrl)
```

This case is not a “quick fix” like the others were. If `model.selectedUrl` is `Nothing` here, we really don’t want to render this `img` at all!

Let’s pull the logic to view the large photo into its own function, and handle the `Maybe` in there by declining to render the `img` at all.

```
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
      ... (everything in between stays the same) ...
      , div [ id "thumbnails", class (sizeToString model.chosenSize) ]
        (List.map (viewThumbnail model.selectedUrl) model.photos)
      , viewLarge model.selectedUrl
    ]

viewLarge : Maybe String -> Html Msg
viewLarge maybeUrl =
  case maybeUrl of
    Nothing ->
      text ""

    Just url ->
      img [ class "large", src (urlPrefix ++ "large/" ++ url) ] []
```

Notice that we’re returning `text ""` if `maybeUrl` is `Nothing`. This “render an empty text node” technique is a quick way to obtain some `Html` that has no visual impact on the page.

Let’s recompile everything with `elm-make --output=elm.js PhotoGroove.elm` and open `index.html` to see our progress.



Figure 4.1 The updated application, with no photos displaying

Excellent! We’ve achieved that peculiar flavor of success where we smile at our desk...while hoping nobody walks by and sees how the previously-working application looks right now.

4.1.2 Resolving Data Dependencies

Now that we’ve resolved our type mismatches, let’s turn to our business logic.

`photoArray` is still based on `initialModel.photos`. This is perfectly reasonable if `initialModel.photos` has a bunch of photos in it...but now that it's hardcoded to `[]`? Not so much. That means `photoArray` will always be empty, which means even once we've loaded some photos to display, our *Surprise Me!* button won't be able to randomly select one.

FIXING SELECTBYINDEX

To fix this, we need to make our `SelectByIndex` logic reference the *current* photos, rather than the hardcoded ones we had at startup. We can do this by deleting `photoArray` and `getPhotoUrl`, and changing the `SelectByIndex` branch of our `update` function like so:

```
SelectByIndex index ->
  let
    newSelectedPhoto : Maybe Photo
    newSelectedPhoto =
      Array.get index (Array.fromList model.photos)

    newSelectedUrl : Maybe String
    newSelectedUrl =
      case newSelectedPhoto of
        Just photo ->
          Just photo.url

        Nothing ->
          Nothing
  in
    ( { model | selectedUrl = newSelectedUrl }, Cmd.none )
```

Our `newSelectedUrl` implementation fits a pattern that comes up fairly often:

1. Run a *case-expression* on a `Maybe`
2. In the `Nothing` branch, evaluate to `Nothing`
3. In the `Just` branch, evaluate to another `Just`

In other words, if we have a `Nothing`, we leave it alone, but if we have a `Just`, we transform it into a different `Just`.

REFACTORING TO USE MAYBE.MAP

The `Maybe.map` function provides a way to perform this transformation more concisely. Let's refactor our `getPhotoUrl` implementation to use it.

```
newSelectedUrl : Maybe String
newSelectedUrl =
  Maybe.map (\photo -> photo.url) newSelectedPhoto
```

You can read this code as "Look at the result of `Array.get index photoArray`. If it's `Nothing`, then return `Nothing`, but if it's `Just photo`, return `Just photo.url` instead."

`Maybe.map` does this by accepting a function which transforms the old `Just` value into the new one. (If the `Maybe` is `Nothing`, this `Just`-transforming function does not get called.)

Table 4.1 illustrates the similarities between `Maybe.map` and our old friend `List.map`.

Table 4.1 Comparing List.map and Maybe.map

List.map	Maybe.map
List.map negate [39] == [-39]	Maybe.map negate (Just 39) == Just -39
List.map negate [5] == [-5]	Maybe.map negate (Just 5) == Just -5
List.map negate [] == []	Maybe.map negate Nothing == Nothing

This code is now significantly more concise than when we used an entire *case-expression* to implement the same logic...*but wait! There's more!*

REFACTORING TO USE .URL

Anytime you see a function that takes a record and immediately return one of its fields—as we're doing with `(\photo -> photo.url)` above—you can take advantage of a special syntax Elm offers for these occasions. If we replace `(\photo -> photo.url)` with `.url` instead, it will do exactly the same thing.

That means we can delete `newSelectedPhoto` and refactor `newSelectedUrl` to this:

```
newSelectedUrl : Maybe String
newSelectedUrl =
    Maybe.map .url (Array.get index (Array.fromList model.photos))
```

TIP You can think of `.url` as “a function which takes a record and returns its `.url` field.”

We've now replaced a *case-expression* with one function call, but the result is not the nicest to read. Fortunately, Elm provides an excellent way to make expressions like this more readable!

USING THE PIPELINE OPERATOR

We can express `newSelectedUrl` as a “pipeline” of sequential operations by using the `|>` operator like so:

```
newSelectedUrl : Maybe String
newSelectedUrl =
    model.photos
        |> Array.fromList
        |> Array.get index
        |> Maybe.map .url
```

This expression is exactly equivalent to writing `Maybe.map .url (Array.get index (Array.fromList model.photos))` except it's expressed in a different style.

Figure 4.2 compares these two ways to write the same expression.

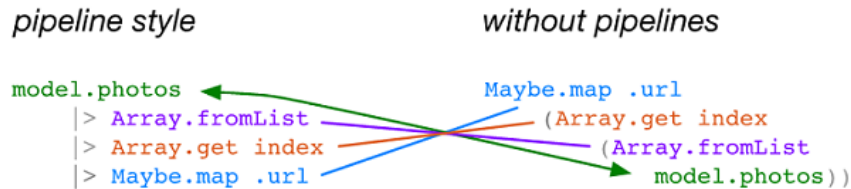


Figure 4.2 The same expression, with and without pipelines

The pipelined code is saying the following:

1. Start with `model.photos`
2. Pass it to `Array.fromList`
3. Pass the previous step's return value as the final argument to `Array.get index`
4. Pass *that* previous step's return value as the final argument to `Maybe.map .url`

Pipelines make it easier to read and to modify sequences of transformations like this. We'll be using them quite often throughout the rest of the book!

NOTE Before Elm's compiler generates JavaScript code, it quietly rewrites pipelines into normal function calls. So `"foo" |> String.reverse` compiles to the same thing as `String.reverse "foo"`. Since they compile to the same thing, there's no performance cost to choosing one over the other. Choose whichever makes for nicer code!

Go ahead and revise our `SelectByIndex` logic to use this pipeline style, like so:

```
SelectByIndex index ->
  let
    newSelectedUrl : Maybe String
    newSelectedUrl =
      model.photos
        |> Array.fromList
        |> Array.get index
        |> Maybe.map .url
  in
    ( { model | selectedUrl = newSelectedUrl }, Cmd.none )
```

INLINING RANDOMPHOTOPICKER

Now we have one remaining problem: `randomPhotoPicker` is still referencing the `photoArray` value we deleted.

Fortunately, it was only using `photoArray` to get its length inside the `SurpriseMe` branch of `update`'s *case-expression*. We can get that same length value using `List.length model.photos` at that point in the code, so we can delete the top-level `randomPhotoPicker` in favor of this:

```
SurpriseMe ->
  let
    randomPhotoPicker =
      Random.int 0 (List.length model.photos - 1)
```

```
in
  ( model, Random.generate SelectByIndex randomPhotoPicker )
```

Great! Now that we’ve installed the requisite package and prepared our data model for the transition, we’re ready to get some HTTP going!

4.2 Fetching Data from a Server

One of our helpful coworkers has set up a simple server endpoint which returns a comma-separated list of photo filenames. The list doesn’t yet contain the metadata that we’ll ultimately need, but it’ll give us a good starting point toward that goal.

Let’s send a `HTTP GET` request to fetch that comma-separated string from a server!

4.2.1 Describing HTTP Requests

We’ll be using the `Http` module to send HTTP requests to our server. Since `Http` is not one of Elm’s core modules, we need to install the `elm-lang/http` package to gain access to it:

```
elm-package install elm-lang/http
```

In Chapter 6 we’ll learn more about package management, but for now let’s keep moving!

MANAGED EFFECTS INSTEAD OF SIDE EFFECTS

In Chapter 3 we saw how generating random numbers in Elm must be done with a `Cmd` rather than a plain function call.

Whereas JavaScript’s `Math.random()` can return a different random number each time you call it, Elm functions must be more consistent. When you pass an Elm function the same arguments, it’s guaranteed return the same value.

There’s another rule that applies to all Elm functions, and this one affects the HTTP requests we’re about to make. The rule is that Elm functions cannot have *side effects*.

DEFINITION An *effect* is an operation that modifies external state. A function which modifies external state when it executes has a *side effect*.

HTTP requests can always modify external state—since even a `GET` request can result in a server changing values in a database—so performing an HTTP request is an effect. This means if we execute a function and it performs an HTTP request, that function has a side effect.

Listing 4.1 shows a few JavaScript functions which consistently return the same value, but which have side effects because they modify external state along the way. None of these could be written as plain Elm functions.

Listing 4.1 JavaScript functions which have side effects

```
function storeStuff() {
  localStorage.stuff = "foo";
```



```

    return 1;
  }

  function editField(object) {
    object.foo = "bar";
    return object;
  }

  var something = 1;

  function editVar() {
    something = 2;
    return 0;
  }

```

- ❶ modifies the contents of `localStorage`
- ❷ modifies the object it receives
- ❸ modifies an external variable

If you look through all the code we've written for *Photo Groove*, you won't find a single function that has a side effect. All effects are performed by the Elm Runtime itself; our code only **describes** which effects to perform, by returning values from `update`.

NOTE Calling `update` does not directly alter any state! All `update` does is return a tuple. If you wanted to, you could call `update` a hundred times in a row, and all it would do is give you a hundred tuples.

This system of *managed effects*, where the Elm Runtime is in charge of performing all effects, means that Elm programs can be written entirely in terms of data transformations. In Chapter 6 we'll see how nice this makes Elm code to test.

This means that in order to perform a HTTP request, we'll do what we did with random number generation in Chapter 3: use a `Cmd` to tell the Elm runtime to perform that effect.

DESCRIBING REQUESTS WITH `HTTP.GETSTRING`

The `Http.send` function returns a `Cmd` representing the HTTP request we want to make.

To configure this request, we pass `Http.send` an `Http.Request` value describing what the request should do. This `Request` value serves a similar purpose to the `Random.Generator` value we passed to `Random.generate` in Chapter 3—except where `Random.Generator` described the random value we wanted, `Request` describes the HTTP request we want.

One way we can obtain a `Request` value is with the `Http.getString` function, like so:

```
Http.getString "http://elm-in-action.com/photos/list"
```

The `getString` function has this type:

```
getString : String -> Request String
```

We pass it a URL string, and it returns a `Request` that asks the Elm Runtime to send a HTTP GET request to that URL. If the request succeeds, we'll end up with a plain `String` containing the response body the server sent back.

TRANSLATING REQUESTS INTO COMMANDS WITH `HTTP.SEND`

Now that we have our `Request String`, we can pass it to `Http.send` to get a `Cmd Msg`. Just like in Chapter 3 with random number generation, once this `Cmd` completes, it will send a `Msg` to `update` telling us what happened.

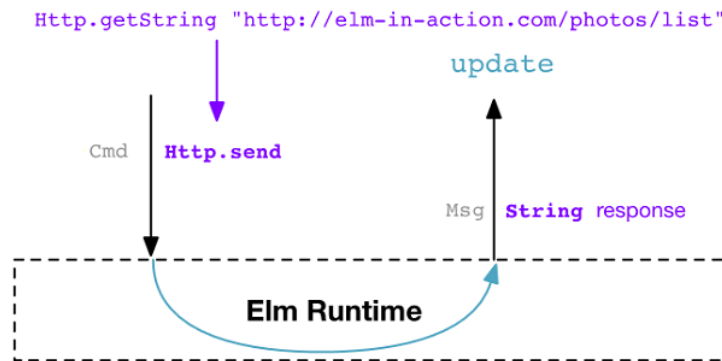


Figure 4.3 The Elm Runtime executing a `Cmd` from `Http.send`

Unlike random number generation, a lot can go wrong with a HTTP request! What if we send the server an invalid URL? Or an unrecognized one? What if the server has been unplugged by a mischievous weasel? If the request fails for any reason, we'll want to know what went wrong so we can inform the user.

`Http.send` produces a value that accounts for this possibility: a `Result`.

4.2.2 Handling HTTP Responses

To get a feel for how `Result` works, let's compare it to its cousin `Maybe`.

In Chapter 3 we saw how `Array.get` returned a `Maybe` in order to account for the possibility of a missing value. The definition of `Maybe` looks like this:

```

type Maybe value
  = Just value
  | Nothing
  
```

Elm has a similar union type for representing the *result* of an operation that can fail, such as performing a HTTP request. It's called `Result`, and it looks like this:

```

type Result errValue okValue
  = Err errValue
  | Ok okValue
  
```

As we can see, `Result` has two type variables:

- `okValue` represents the type of value we'll get if the operation succeeded. For example, if we are trying to obtain a `String` from the server, then `okValue` would be `String`.
- `errValue` represents the type of error we'll get if the operation failed. In our case, `errValue` will refer to a union type that enumerates various mishaps that can befall HTTP requests. That union type is called `Http.Error`.

Similarly to how `Maybe` requires that we write logic to handle both the `Just` and `Nothing` cases, so too does `Result` require that we handle both the `Ok` and `Err` cases.

HANDLING THE RESULT

Let's start by importing the `Http` module we're about to begin using.

```
import Http
```

Next let's add a `Msg` type constructor to handle the `Result` value we'll get from `Http.send`:

```
type Msg
  = SelectByUrl String
  | SelectByIndex Int
  | SurpriseMe
  | SetSize ThumbnailSize
  | LoadPhotos (Result Http.Error String)
```

We'll learn more about that `Http.Error` type in a bit. For now, let's add a branch to our `update` function's *case-expression* for this new `Msg`. We'll start with something like this:

```
LoadPhotos result ->
  case result of
    Ok responseStr ->
      ...translate responseStr into a List of Photos for our Model...

    Err httpError ->
      ...record an error message here...
```

With that basic skeleton in place, we can implement these two cases one at a time. We'll start with the `Ok responseStr` case, where the server gave us a valid response and all is well.

READING PHOTO FILENAMES WITH `STRING.SPLIT`

Remember earlier when we mentioned how we've got a server endpoint which returns a comma-separated list of photo filenames? If not, here it is again:

One of our helpful coworkers has set up a simple server endpoint which returns a comma-separated list of photo filenames.

—This book, earlier.

Sure enough, if we visit this endpoint at <http://elm-in-action.com/photos/list> we can see the list of filenames. It's a plain old string:

```
"1.jpeg,2.jpeg,3.jpeg,4.jpeg"
```

When we wrote `Ok responseStr` a moment ago, that `responseStr` value will refer to exactly this string! Now we can split `responseStr` into a list of individual filenames using Elm's `String.split` function. It has this type:

```
split : String -> String -> List String
```

We give `String.split` a separator (in this case a comma) and a string to split, and it gives us back a list of all the strings it found between those separators.

Table 4.2 shows what `String.split ","` will do to the string we get from the server.

Table 4.2 Calling `String.split ","` on a string

<code>stringFromServer</code>	<code>String.split "," stringFromServer</code>
"1.jpeg,2.jpeg,3.jpeg,4.jpeg"	["1.jpeg", "2.jpeg", "3.jpeg", "4.jpeg"]

Once we have this list of filename strings, we can use it to set the `photos : List Photo` field in our `Model` like in Listing 4.2.

Listing 4.2 Incorporating `responseStr` into the Model

```
Ok responseStr ->
  let
    urls =
      String.split "," responseStr ❶

    photos =
      List.map (\url -> { url = url }) urls ❷
  in
    ( { model | photos = photos }, Cmd.none ) ❸
```

- ❶ Split the String into a List String
- ❷ Translate the List String into a List Photo
- ❸ Set the photos in the model

Now when we successfully receive a `String` response from the server, we'll translate it into a list of `Photo` records and store them in our `Model`.

USING TYPE ALIASES TO CREATE RECORDS

Declaring `type alias Photo = { url : String }` does more than giving us a `Photo` type we can use in type annotations. It also gives us a convenience function whose job is to build `Photo` record instances. This function is also called `Photo`. Here it is in action:

```
Photo "1.jpeg" == { url = "1.jpeg" }
```

This also works with record type aliases involving multiple fields, like the one for `Model`:

```
type alias Model =
  { photos : List Photo
  , selectedUrl : String
  , chosenSize : ThumbnailSize
  }
```

This declaration gives us a convenience function called `Model` which builds a record and returns it. Since this record has three fields where `Photo` had only one, the `Model` function accepts three arguments instead of one. The type of the `Model` function looks like this:

```
Model : List Photo -> String -> ThumbnailSize -> Model
```

The order of arguments matches the order of the fields in the `type alias` declaration. So if you were to move the `photos : List Photo` declaration to the end of the `type alias`, then the `Model` function would look like this instead:

```
Model : String -> ThumbnailSize -> List Photo -> Model
```

We can use this knowledge to perform a quick refactor of our `Photo` construction:

```
Old: List.map (\url -> { url = url }) urls
New: List.map Photo urls
```

Lovely!

HANDLING ERRORS (OR NOT)

What about when things don't go as planned? We'll add some nice error handling later, but first we want to get something up and running. Elm won't let us forget to handle the error case—if we omitted the `Err` branch, we'd get a `MISSING PATTERNS` compile error!—but the compiler won't stop us from choosing to do nothing when we receive an `Err`.

```
Err httpError ->
  ( model, Cmd.none )
```

Since we aren't using our `httpError` value, let's rename it to an underscore:

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

The underscore parameter

The underscore is a special parameter name whose purpose is to signify that there is a parameter here, but we're choosing not to use it. Unlike a normal parameter, attempting to reference `_` in our logic would be a compile error.

One handy feature of `_` is that you can use it multiple times in the same parameter list. For example:

```
functionThatTakesThreeArguments _ _ _ =
```

```
  "I ignore all three of my arguments and return this string!"
```

You can use `_` either in function parameter lists or in case-expression branches.

Later on we'll circle back to this branch of our *case-expression* and use it to show the user a nice error message when an HTTP error happens.

PATTERN MATCHING

Before we move on, let's pause for a tasty refactor. If you zoom out a bit, you'll notice we have a *case-expression* nested directly inside another *case-expression*.

```
case msg of
  ...

  LoadPhotos result ->
    case result of
      Ok responseStr ->
        ...

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

In situations like this, we can use concise *pattern matching* to express the same logic:

```
case msg of
  ...

  LoadPhotos (Ok responseStr) ->
    ...

  LoadPhotos (Err _) ->
    ( model, Cmd.none )
```

DEFINITION *Pattern matching* is a way of destructuring values based on how their containers look. In the example above, if we have a `LoadPhotos` containing an `Ok` containing a value, that value will go into a variable called `responseStr`.

This refactored code is equivalent to what we had before. The difference is that now each of our branches is expressing two conditions at once:

- The `LoadPhotos (Ok responseStr)` branch runs if `msg` is a `LoadPhotos` type constructor which contains an `Ok` value.
- The `LoadPhotos (Err _)` branch runs if `msg` is a `LoadPhotos` type constructor and it contains an `Err` value.

You can nest patterns like these as much as you want, even assembling elaborate creations like `NestAllTheThings (Just (Ok (Listen (This (IsReally "great" _ _)))))` - but try not to overdo it.

We'll encounter other pattern matching techniques as we progress through the book.

4.2.3 Sending HTTP Requests

At this point we've assembled all the ingredients necessary to launch our HTTP rocketship in the direction of our server:

- The `Request String` returned by `Http.getString "http://elm-in-action.com/photos/list"`
- The `LoadPhotos` message that will handle the response

The `Http.send` function will wire these up for us! It has this type:

```
send : (Result Http.Error val -> msg) -> Request val -> Cmd msg
```

`Http.send` has two type variables: `msg` and `val`. We can infer what they'll become in our case:

- Since we're passing a `Request String`, the `val` type variable must become `String`
- Since we need it to return a `Cmd Msg`, the `msg` type variable must become `Msg`

Knowing this, we can tell that the first argument to `send` is a function that looks like this:

```
(Result Http.Error String -> Msg)
```

This means we could declare a new value called `initialCmd` using this call to `Http.send`:

```
initialCmd : Cmd Msg
initialCmd =
  Http.send
    (\result -> LoadPhotos result)
    (Http.getString "http://elm-in-action.com/photos/list")
```

We'll use this `initialCmd` value to run our HTTP request when the program starts up.

This will compile, but we can simplify it! Back in Section 2.2.1 of Chapter 2, we noted that an anonymous function like `(\foo -> bar baz foo)` can always be rewritten as `(bar baz)` by itself. This means we can replace `(\result -> LoadPhotos result)` with `LoadPhotos` like so:

```
initialCmd : Cmd Msg
initialCmd =
  Http.send LoadPhotos
    (Http.getString "http://elm-in-action.com/photos/list")
```

USING PIPELINE STYLE FOR INITIALCMD

By the way, remember how earlier we refactored `newSelectedUrl` to use a pipeline style instead of multiple constants? We can use the same technique to express `initialCmd` in terms of the steps needed to build it up:

```
initialCmd : Cmd Msg
initialCmd =
  "http://elm-in-action.com/photos/list"
    |> Http.getString
    |> Http.send LoadPhotos
```

Nice! Let's add this version of `initialCmd` to `PhotoGroove.elm` right above our main declaration.

RUNNING A COMMAND ON INIT

Now we have a `Cmd` to run, but we want to kick this one off a bit differently than last time. We ran our random number generation command from `update` in response to a user click, but we want to run this command right when the program starts up.

We can do that by updating `main` to use `initialCmd` instead of `Cmd.none` on `init`, like so:

```
main : Program Never Model Msg
main =
  Html.program
    { init = ( initialModel, initialCmd )
    , view = view
    , update = update
    , subscriptions = \_ -> Sub.none ❶
    }
```

❶ We don't use this argument, so name it `_`

While we're at it, we also updated `subscriptions` to use `_` for its anonymous function's argument, since that argument never gets used.

TIP An anonymous function which ignores its argument (like the `_ ->` above) looks like a hockey stick.

Figure 4.4 illustrates how our revised `init` kicks off the sequence of events leading up to our `update` function receiving a `LoadPhotos` message.

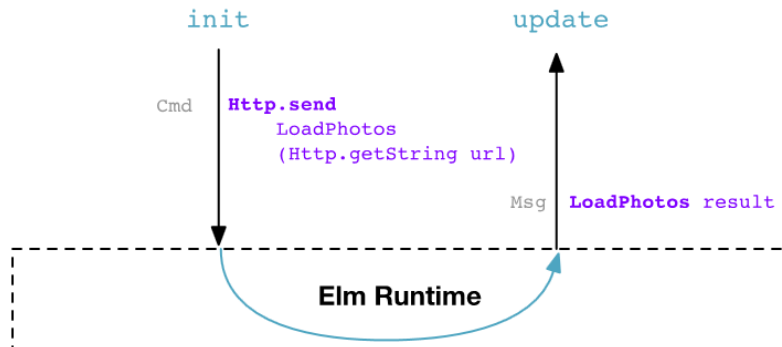


Figure 4.4 Running a `LoadPhotos` command on `init`

Now let's recompile and open `index.html`, so we can bask in the glory of our new server-loaded photos!

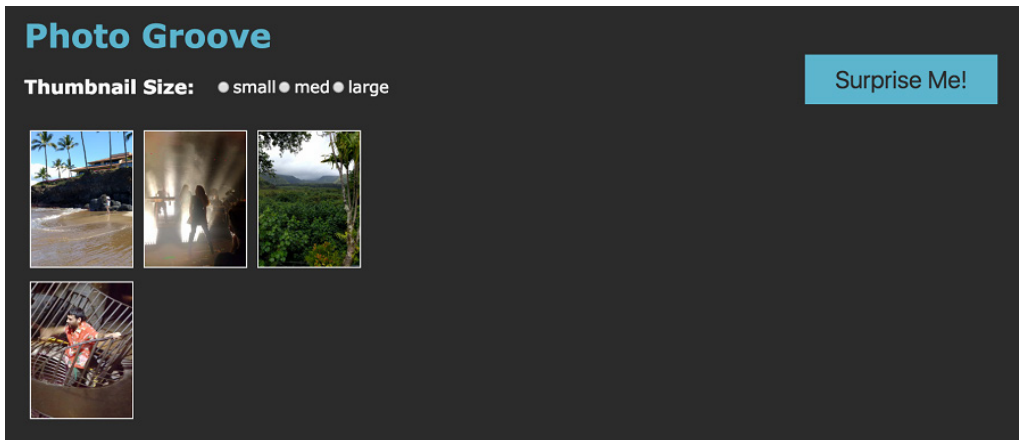


Figure 4.5 The updated application, with an additional photo loaded

Excellent! However, we've lost something along the way: we no longer select the first photo like we did before.

SELECTING AN INITIAL PHOTO USING LIST.HEAD

We'd like to select the first URL that comes back from the server, whatever that URL may be. The `List.head` function is perfect for this! Its type looks like this:

```
head : List elem -> Maybe elem
```

`List.head someList` and `Array.get 0 someArray` serve essentially the same purpose: returning `Nothing` if run on an empty collection, or `Just whateverElementWasFirst` if the collection had anything in it. This means we can grab the first URL out of the list like so:

```
List.head urls
```

Since we're going to set `selectedUrl` to this value, and `selectedUrl` is already a `Maybe String`, we don't need run a *case-expression* on this `Maybe String` to do any special handling. We can drop it in and let whoever works with `selectedUrl` handle the `Nothing` case.

Let's expand the record update in the `LoadPhotos (Ok responseStr)` branch to set not only the `photos` field but the `selectedUrl` field as well:

```
LoadPhotos (Ok responseStr) ->
  let
    urls =
      String.split "," responseStr

    photos =
      List.map Photo urls
  in
    ( { model
```

```

    | photos = photos
    , selectedUrl = List.head urls
  }
  , Cmd.none
)

```

Now we can try it out and see how this looks. Recompile and...

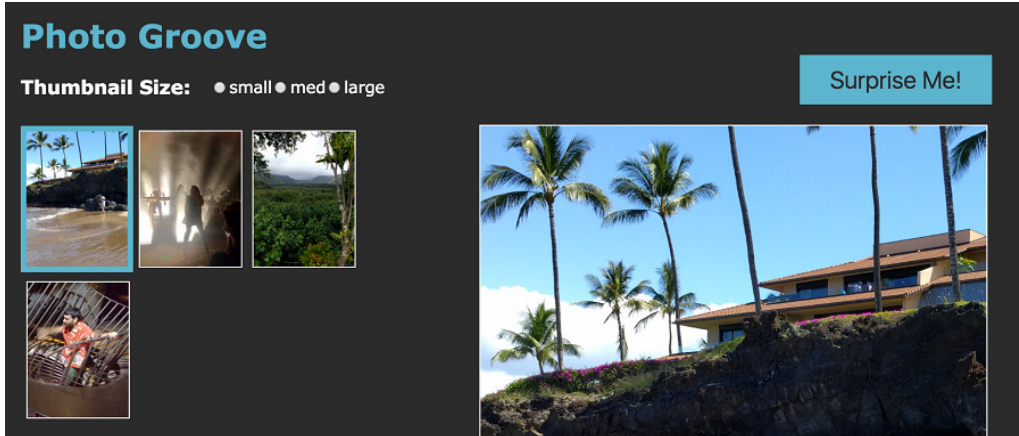


Figure 4.6 Selecting the first loaded photo by default.

Beautiful! Now that we're loading photos from the server, we're ready to take the final step: obtaining the metadata and complete URLs from the server rather than just the filenames.

Now that we've gotten things working in the case where the response successfully comes back, let's revisit the case where everything is not all sunshine and kittens. It's time for some proper error handling!

4.2.4 Gracefully Handling Errors

Let's make it so that if there's an error, we render a helpful message for our users. To do this, we'll need to make two changes:

1. Have `update` set this error message when it receives a `ReportError` message.
2. Have `view` render this error message if it's present.

Let's start with the change to `update`:

```

LoadPhotos (Err _) ->
  ( { model
    | loadingError = Just "Error! (Try turning it off and on again?)"
    }
  , Cmd.none
  )

```

NOTE We need the `Just` here because we're storing `loadingError` in our `Model` as a `Maybe String`, whereas `ReportError` holds a plain `String`.

Next let's introduce a brand-new function which renders the error message, and then modify `Html.program` to use it instead of using `view` directly:

```
viewOnError : Model -> Html Msg
viewOnError model =
  case model.loadingError of
    Nothing ->
      view model

    Just errorMessage ->
      div [ class "error-message" ]
        [ h1 [] [ text "Photo Groove" ]
        , p [] [ text errorMessage ]
        ]

main : Program Never Model Msg
main =
  Html.program
    { init = ( initialModel, initialCmd )
    , view = viewOnError
```

Whoa! Can we do that? Can we pass a function other than `view` to `Html.program` as the view function to use?

Well, sure! All `Html.program` needs there is a function of type `Model -> Html Msg`. As long as we give it a function of that type, `Html.program` could care less what that function is called.

We've done several refactors where we split smaller functions out of `view`, but this is the first time we've wrapped `view` in another function. And it was no big deal! One of the nice things about working with plain old Elm functions all the time is that they're easily interchangeable as long as the types line up.

VERIFYING THAT ERROR HANDLING WORKS

Let's start by making sure the error handling works properly. Change `initialCmd` to fetch data from an invalid URL:

```
initialCmd : Cmd Msg
initialCmd =
  "http://elm-in-action.com/breakfast-burritos/list"
  |> Http.getString
  |> Task.perform handleLoadFailure handleLoadSuccess
```

Now when you recompile and open `index.html`, you'll see our new error message:

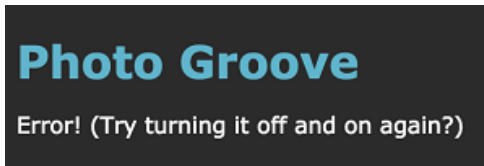


Figure 4.7 Displaying an error as a status message

Now let's change `initialCmd` back to `"http://elm-in-action.com/photos/list"` and press on toward receiving that photo metadata!

4.3 Decoding JSON

Browsers can talk to servers using many different formats. JSON, XML, RPC, BBQ...the list goes on. Regardless of the format, when the browser sends off a HTTP request to a server, the response comes back as a raw string. It's up to the program running in the browser to decode that raw string into a useful representation.

The server with the metadata will be sending us that information via JSON. This means we'll need some way to decode a JSON string into something more useful!

4.3.1 Decoding JSON Strings into Results

Plenty of things can go wrong during the process of decoding JSON. Suppose the server messes up and sends us XML instead of JSON. Our JSON-decoding logic will certainly fail! Or suppose we get back a response that's valid JSON, but a required field is missing. What then?

THE DECODESTRING FUNCTION

The `Json.Decode.decodeString` function returns a `Result`. Its complete type is:

```
decodeString : Decoder val -> String -> Result String val
```

Notice that the `Result` we get back from `decodeString` has one concrete type—namely that `errValue` is always a `String`—and one that matches the type of the given decoder's type variable. Table 4.3 shows how the decoder passed to `decodeString` affects its return type.

Table 4.3 How the Decoder passed to `decodeString` affects its return type

Decoder passed in	<code>decodeString</code> returns	Example success value	Example failure value
Decoder <code>Bool</code>	Result <code>String Bool</code>	Ok <code>True</code>	Err <code>"nope"</code>
Decoder <code>String</code>	Result <code>String String</code>	Ok <code>"Win!"</code>	Err <code>"nah"</code>
Decoder <code>(List Int)</code>	Result <code>String (List Int)</code>	Ok <code>[1, 2, 3]</code>	Err <code>"argh!"</code>

DECODING PRIMITIVES

The `Json.Decode` module has a `Decoder Bool` called `bool`, which translates a JSON boolean string (either `"true"` or `"false"`) into an Elm `Bool` (`True` or `False`). Let's try it in `elm-repl`!

Listing 4.1 Using `decodeString bool`

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

> decodeString bool "true"
Ok True : Result String Bool

> decodeString bool "false"
Ok False : Result String Bool

> decodeString bool "42"
Err "Expecting a Bool but instead got: 42" : Result String Bool

> decodeString bool "@&!*/%?"
Err "Given an invalid JSON: Unexpected token @" : Result String Bool
```

NOTE `elm-repl` prints types in fully-qualified style to avoid ambiguity, meaning it always prints `Result.Result` instead of `Result`. These examples will use the unqualified `Result` for brevity, which means they won't have the quite same output as what you'll see in your own `elm-repl`.

Besides `bool`, the `Json.Decode` module offers other primitive decoders like `string`, `int`, and `float`. They work similarly to `bool`, as we can see in `elm-repl`:

Listing 4.2 Using `decodeString` with `int`, `float`, and `string`

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

> decodeString float "3.33"
Ok 3.33 : Result String Float

    > decodeString string "\"backslashes escape quotation marks\""
Ok "backslashes escape quotation marks" : Result String String

> decodeString int "76"
Ok 76 : Result String Int

> decodeString int "3.33"
Err "Expecting an Int but instead got: 3.33" : Result String Int
```

The only primitive decoders are `bool`, `int`, `float`, `string`, and `null`, since JavaScript's `undefined` is not allowed in valid JSON. As we will soon see, there are more popular ways of handling `null` than with this primitive decoder.

4.3.2 Decoding JSON Collections

We've now seen how to decode primitives like booleans, integers, and strings from JSON into their Elm counterparts. This is a good start, but JSON also supports arrays and objects, and we'll need to decode both of these types in order to receive our list of photos with metadata.

DECODING JSON ARRAYS INTO LISTS

Suppose we have the JSON string `"[true, true, false]"`. To decode a list of booleans from this JSON array, we can write `list bool`. This will give us a `Decoder (List Bool)` value.

Here's how `Json.Decode.bool` and `Json.Decode.list` compare:

```
bool : Decoder Bool
list : Decoder value -> Decoder (List value)
```

Whereas `bool` is a decoder, **`list` is a function** that takes a decoder and returns a new one. We can use it in `elm-repl` to make decoders for lists of primitives, or even lists of lists!

Listing 4.2 Using `Json.Decode.list`

```
> import Json.Decode exposing (list, bool, string, int)

> list
<function> : Decoder a -> Decoder (List a) ❶

> list bool
Decoder (List Bool)

> list string
Decoder (List String)

> list (list int)
Decoder (List (List Int))
```

❶ `elm-repl` chooses “a” for the type variable we called “value”

DECODING OBJECTS

The simplest way to decode an object is with the `field` function. Suppose we write this:

```
decoder : Decoder String
decoder =
  field "email" string
```

When this decoder runs, it performs three checks:

1. Are we decoding an Object?
2. If so, does that Object have a field called `email`?
3. If so, is the Object's `email` field a String?

If all three are true, then decoding succeeds with the value of the Object's `email` field.

Table 4.4 shows how this decoder would work on a variety of inputs.

Table 4.4. Decoding objects with field decoders

Decoder	JSON	Result
field "email" string	5	Err "Expected object, got 5"
	{"email": 5}	Err "Expected string for 'email', got 5"
	{"email": "cate@nolf.com"}	Ok "cate@nolf.com"

DECODING MULTIPLE FIELDS

Building a decoder for a single `field` is all well and good, but typically when decoding objects, we care about more than one of their fields. How do we do that?

The simplest is with a function like `map2`, which we can see in Table 4.5.

Table 4.5. Decoding objects with field decoders

Decoder	JSON	Result
<code>map2</code> <code>(\x y -> (x, y))</code> <code>(field "x" int)</code> <code>(field "y" int)</code>	{"width": 5}	Err "Missing 'y' field"
	{"x": 5, "y": null}	Err "Expected int for 'y', got null"
	{"x": 5, "y": 12}	Ok (5, 12)

DECODING MANY FIELDS

The photo information we'll be getting back from our server will be in the form of JSON which looks like this:

```
{"url": "1.jpeg", "size": 36, "title": "Beachside"}
```

This is an object with three fields: two strings and one int. Let's update our `Photo` type alias to reflect this:

```
type alias Photo =
  { url : String
  , size : Int
  , title : String
  }
```

We could decode this using that technique we just learned, with one slight difference. We'd have to use `map3` instead of `map2`, because we have three fields instead of two.

```
photoDecoder : Decoder Photo
photoDecoder =
  map3
    (\url size title -> { url = url, size = size, title = title })
    (field "url" string)
    (field "size" int)
    (field "title" string)
```

How far can this approach take us? If we added a fourth field, we'd change `map3` to `map4`. The `Json.Decode` module also includes `map5`, `map6`, `map7`, and `map8`, but `map8` is as high as it goes.

From there we can either combine the decoders we've already used...or we can introduce a library designed for larger-scale JSON decoding!

PIPELINE DECODING

The `Json.Decode.Pipeline` is designed to make life easier when decoding large objects. It comes from a popular third-party package called `NoRedInk/elm-decode-pipeline` - so let's install it real quick before we proceed:

```
elm-package install NoRedInk/elm-decode-pipeline
```

Let's also write a decoder for it. We'll do that by deleting the following import:

```
import Html.Attributes exposing (..)
```

...and then adding all of this to the end of our imports list:

```
import Html.Attributes exposing (id, class, classlist, src, name, type_, title)
import Json.Decode exposing (string, int, list, Decoder)
import Json.Decode.Pipeline exposing (decode, required, optional)
```

Then the coast is clear to replace `photoDecoder` with this:

Listing 4.2 photoDecoder

```
photoDecoder : Decoder Photo
photoDecoder =
  decode buildPhoto
    |> required "url" string
    |> required "size" int
    |> optional "title" string "(untitled)"

buildPhoto : String -> Int -> String -> Photo
buildPhoto url size title =
  { url = url, size = size, title = title }
```

- ❶ Pass decoded values to the `buildPhoto` function
- ❷ "url" is required, and must be a string
- ❸ "size" is required, and must be an integer
- ❹ "title" is optional, and defaults to "(untitled)"

NOTE We changed our `import Html.Attributes` line because if it were still `exposing (..)`, then we'd be exposing two functions called `required`: both `Json.Decode.Pipeline.required` and `Html.Attributes.required`. Referencing a value that's exposed by two modules at once is an error!

Let's break down what's happening here.

1. `decode buildPhoto` begins the pipeline. It says that our decoder will decode the arguments to `buildPhoto`, one by one, and ultimately the whole decoder will succeed

unless any of the steps in this pipeline fails. Since `buildPhoto` accepts three arguments, we'll need three pipeline steps after this. otherwise, the compiler will give an error.

2. `required "url" string` says that we need what we're decoding to be a JSON object with the `String` field "url". We're also saying that if decoding succeeds, we should use this first result as the first argument to `buildPhoto`. Decoding could fail here, because either the "url" field was missing, or because the field was present...but was not a `String`.
3. `required "size" int` does the same thing `required "url" string` except that it decodes to an integer instead of a string.
4. `optional "title" string "(untitled)"` - this is similar to the required steps, but with one important difference: in this example, if the "title" field were either missing or null, instead of failing decoding, this decoder will default on that final argument—that is, the title string would default to `"(untitled)"`.

Figure 4.8. shows how the `buildPhoto` arguments and `photoDecoder` arguments match up.

<code>photoDecoder : Decoder</code>	<code>Photo</code>	
<code>photoDecoder =</code>		
<code>decode buildPhoto</code>		<code>buildPhoto :</code>
<code> > required "url" string</code>		<code>String</code>
<code> > required "size" int</code>		<code>-> Int</code>
<code> > optional "title" string "(untitled)"</code>		<code>-> String</code>
		<code>-> Photo</code>

Figure 4.8 The relationship between `photoDecoder` and `buildPhoto`

By the way, notice anything familiar about what `buildPhoto` does?

All it does is take one argument for each of the fields in `Photo`, and then assign them uncritically without altering them in any way. We already have a function that does this! It's the `Photo` function, which we got it for free because we defined the `Photo` type alias.

Let's delete `buildPhoto` and replace `photoDecoder = decode buildPhoto` with this:

```
photoDecoder =
  decode Photo
```

Much easier!

WARNING Reordering any function's arguments can lead to unpleasant surprises. Since reordering the fields in the `Model` type alias has the consequence of reordering the `Model` function's arguments, you should be exactly as careful when reordering a type alias as you would be when reordering any function's arguments!

4.3.3 Decoding JSON HTTP Responses

We've already seen how we can use `Http.getString` to obtain a `String` from a server, and how we can use `decodeString` and a `Decoder` to translate a `String` into a list of `Photo` records for our `Model`.

Although we could use `Http.getString` and `decodeString` to populate our `Model` in this way, there is another function in the `Http` module which will take care of both for us.

HTTP.GET

The `Http.get` function requests data from a server and then decodes it. Here is how the types of `Http.getString` and `Http.get` match up:

```
getString : String -> Request String
get       : Decoder value -> String -> Request value
```

Comparing types like this suggests how these functions are similar and how they differ.

They both accept a URL string and return a `Request`. However, where `getString` takes no other arguments and produces a `String` on success, `get` additionally accepts a `Decoder` value, and on success produces a `value` instead of a `String`.

As you might expect, if we give `Http.get` a decoder of `(list int)` and the response it gets back is the JSON payload `"[1, 2, 3]"`, then `Http.get` will successfully decode that into `Ok (List Int)`. If decoding fails, we will instead get `Err UnexpectedPayload`.

What's `UnexpectedPayload`? It's a type constructor for that `Http.Error` type we said we'd get back to! (Sure enough, here we are getting back to it.) `Http.Error` is a union type which describes various ways a HTTP request can fail. It looks like this:

```
type Error
= BadUrl String
| Timeout
| NetworkError
| BadStatus (Response String)
| BadPayload String (Response String)
```

Since it's a union type, we can run a *case-expression* on any `Http.Error` value—with different branches for `Timeout`, `NetworkError`, and so on—to do custom error handling based on what went wrong.

TIP The lower-level `Http.request` function lets you customize requests in more depth, including performing error handling based on raw status code.

Since we want to decode the JSON from our server into a `List Photo`, and we have a `Decoder Photo`, we can use `Json.Decode.list` to end up with the information we want. Let's change `initialCmd` to do just that while it references our shiny new URL:

```
initialCmd : Cmd Msg
initialCmd =
  list photoDecoder
```

```
> Http.get "http://elm-in-action.com/photos/list.json"
> Http.send LoadPhotos
```

This means we'll be sending a `List Photo` to `LoadPhotos` instead of a `String`, so we'll need to update its definition to match:

```
type Msg
= SelectByUrl String
| SelectByIndex Int
| SurpriseMe
| SetSize ThumbnailSize
| LoadPhotos (Result Http.Error (List Photo))
```

This lets us simplify the `LoadPhotos (Ok ...)` branch of `update`'s *case-expression* quite a bit! Having direct access to `photos` means we no longer need to build that value up using a *let-expression*. We can also use our new friends `Maybe.map` and `.url` to determine `selectedUrl` right on the same line!

```
LoadPhotos (Ok photos) ->
  ( { model
    | photos = photos
    , selectedUrl = Maybe.map .url (List.head photos)
  }
  , Cmd.none
  )
```

RENDERING THE METADATA

Now all that remains is to have our view render the new caption and download size metadata!

We can do this by adding one line to `viewThumbnail`:

```
viewThumbnail selectedUrl thumbnail =
  img
    [ src (urlPrefix ++ thumbnail.url)
    , title (thumbnail.title ++ " [" ++ toString thumbnail.size ++ " KB]")
    , classList [ ( "selected", selectedUrl == Just thumbnail.url ) ]
    , onClick (SelectByUrl thumbnail.url)
    ]
  []
```

Great! At this point everything should compile, and we can open `index.html` to see the result:



Figure 4.9 The final application

Now we're reading our list of photos from the server rather than hardcoding them, and we've given ourselves a nice foundation on which to build an even richer application!

4.4 Summary

We learned quite a few things in the course of making Photo Groove talk to servers!

- A `Decoder` can validate and translate JSON into an Elm value.
- The `Json.Decode` module provides primitive decoders like `float`, `int`, and `string`.
- The `Json.Decode.list` function turns a `Decoder Bool` into a `Decoder (List Bool)`.
- The `Json.Decode.Pipeline` module offers functions to decode objects in pipeline style.
- *Pattern matching* lets us trade nested *case-expressions* for longer branch conditions.
- A `Result` is either `Ok okValue` in case of success, or `Err errValue` in case of failure.
- `String.split` splits a string around a given separator, resulting in a list of strings.
- A `Request` describes a request we want to make. `Http.send` turns it into a `Cmd`.
- `Http.getString` requests a plain `String` from the server at the given URL.
- `Http.get` works like `Http.getString`, except it runs the given `Decoder` on the result.
- The `init` field passed to `Html.program` lets us specify a `Cmd` to run on startup.

We also saw how the pipeline operator `(|>)` lets us write expressions by starting with a value and then running a series of transformations on it. Table 4.6 shows an example of this.

Table 4.6 The same expression with and without pipelines

Pipeline Style	Without Pipelines
<pre>model.photos > Array.fromList > Array.get index</pre>	<pre>Array.get index (Array.fromList model.photos)</pre>

Now that we've gotten *Photo Groove* talking to a server, we'll get it talking to JavaScript. This will let us tap into the enormous ecosystem of JavaScript libraries out there. Let's see what that can get us!

Listing 3.7 The complete PhotoGroove.elm

```
module PhotoGroove exposing (..)

import Html exposing (..)
import Html.Events exposing (onClick)
import Array exposing (Array)
import Random
import Http
import Html.Attributes exposing (id, class, classlist, src, name, type_, title)
import Json.Decode exposing (string, int, list, Decoder)
import Json.Decode.Pipeline exposing (decode, required, optional)

photoDecoder : Decoder Photo
photoDecoder =
  decode Photo
    |> required "url" string
    |> required "size" int
    |> optional "title" string "(untitled)"

urlPrefix : String
urlPrefix =
  "http://elm-in-action.com/"

type ThumbnailSize
  = Small
  | Medium
  | Large

view : Model -> Html Msg
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
    , button
      [ onClick SurpriseMe ]
      [ text "Surprise Me!" ]
    , h3 [] [ text "Thumbnail Size:" ]
    , div [ id "choose-size" ]
      (List.map viewSizeChooser [ Small, Medium, Large ])
    , div [ id "thumbnails", class (sizeToString model.chosenSize) ]
      (List.map (viewThumbnail model.selectedUrl) model.photos)
```

```

        , viewLarge model.selectedUrl
    ]

viewLarge : Maybe String -> Html Msg
viewLarge maybeUrl =
    case maybeUrl of
        Nothing ->
            text ""

        Just url ->
            img [ class "large", src (urlPrefix ++ "large/" ++ url) ] []

viewThumbnail : Maybe String -> Photo -> Html Msg
viewThumbnail selectedUrl thumbnail =
    img
        [ src (urlPrefix ++ thumbnail.url)
        , title (thumbnail.title ++ " [" ++ toString thumbnail.size ++ " KB]")
        , classlist [ ( "selected", selectedUrl == Just thumbnail.url ) ]
        , onClick (SelectByUrl thumbnail.url)
        ]
    []

viewSizeChooser : ThumbnailSize -> Html Msg
viewSizeChooser size =
    label []
        [ input [ type_ "radio", name "size", onClick (SetSize size) ] []
        , text (sizeToString size)
        ]

sizeToString : ThumbnailSize -> String
sizeToString size =
    case size of
        Small ->
            "small"

        Medium ->
            "med"

        Large ->
            "large"

type alias Photo =
    { url : String
    , size : Int
    , title : String
    }

type alias Model =
    { photos : List Photo
    , selectedUrl : Maybe String
    , loadingError : Maybe String
    , chosenSize : ThumbnailSize
    }

```

```

    }

initialModel : Model
initialModel =
    { photos = []
    , selectedUrl = Nothing
    , loadingError = Nothing
    , chosenSize = Medium
    }

photoArray : Array Photo
photoArray =
    Array.fromList initialModel.photos

getPhotoUrl : Int -> Maybe String
getPhotoUrl index =
    case Array.get index photoArray of
        Just photo ->
            Just photo.url

        Nothing ->
            Nothing

type Msg
    = SelectByUrl String
    | SelectByIndex Int
    | SurpriseMe
    | SetSize ThumbnailSize
    | LoadPhotos (Result Http.Error (List Photo))

randomPhotoPicker : Random.Generator Int
randomPhotoPicker =
    Random.int 0 (Array.length photoArray - 1)

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        SelectByIndex index ->
            let
                newSelectedUrl : Maybe String
                newSelectedUrl =
                    model.photos
                    |> Array.fromList
                    |> Array.get index
                    |> Maybe.map .url
            in
                ( { model | selectedUrl = newSelectedUrl }, Cmd.none )

        SelectByUrl url ->
            ( { model | selectedUrl = Just url }, Cmd.none )

        SurpriseMe ->

```

```

    let
      randomPhotoPicker =
        Random.int 0 (List.length model.photos - 1)
    in
      ( model, Random.generate SelectByIndex randomPhotoPicker )

SetSize size ->
  ( { model | chosenSize = size }, Cmd.none )

LoadPhotos (Ok photos) ->
  ( { model
    | photos = photos
    , selectedUrl = Maybe.map .url (List.head photos)
    }
  , Cmd.none
  )

LoadPhotos (Err _) ->
  ( { model
    | loadingError = Just "Error! (Try turning it off and on again?)"
    }
  , Cmd.none
  )

initialCmd : Cmd Msg
initialCmd =
  list photoDecoder
  |> Http.get "http://elm-in-action.com/photos/list.json"
  |> Http.send LoadPhotos

viewOnError : Model -> Html Msg
viewOnError model =
  case model.loadingError of
    Nothing ->
      view model

    Just errorMessage ->
      div [ class "error-message" ]
        [ h1 [] [ text "Photo Groove" ]
        , p [] [ text errorMessage ]
        ]

main : Program Never Model Msg
main =
  Html.program
  { init = ( initialModel, initialCmd )
  , view = viewOnError
  , update = update
  , subscriptions = (\_ -> Sub.none)
  }

```


5

Talking to JavaScript

This chapter covers

- Rendering Custom Elements
- Sending data to JavaScript
- Receiving data from JavaScript
- Initializing our Elm application using data from JavaScript

Now we've gotten *Photo Groove* loading photos from a server. It's looking better and better! However, our manager has a concern. "We have the photos, and they certainly look sharp. But where is the *groove*? This is Photo Groove, not Photo Browse! You know what we need? *Filters*. You know, the kind that make normal photos look all wacky and messed-up? Those!"

Sounds groovy.

We collaborate with our team's visual designer to create a mockup of the filtering feature. The final design calls for a status bar to report on the filtering process, and sliders to control the filters' settings. And not just any sliders! Our designer wants each to have the specific look and feel of a Paper Slider, a Web Component built on Google's Material Design guidelines.

As luck would have it, we won't have to code either the filter effects or the sliders from scratch. They're both available as open-source libraries, but there's a catch: these libraries are written in JavaScript, not Elm.

Fortunately, Elm applications are not limited to using Elm libraries alone! In this chapter we'll expand Photo Groove to include a filtering feature. We'll add sliders which let users apply filters in varying degrees to the large photo. We'll create a status bar that displays information from the JavaScript libraries. Along the way we'll learn two different ways to incorporate JavaScript code into an Elm application: *custom elements* and *ports*.

Here we go!

5.1 Using Custom Elements

We'll start with the sliders. We want to create three sliders in total: **Hue**, **Ripple**, and **Noise**. Adjusting each value will transform how the large photo appears to the end user.

Figure 5.1 shows how the sliders fit into what we'll be building toward in this chapter.

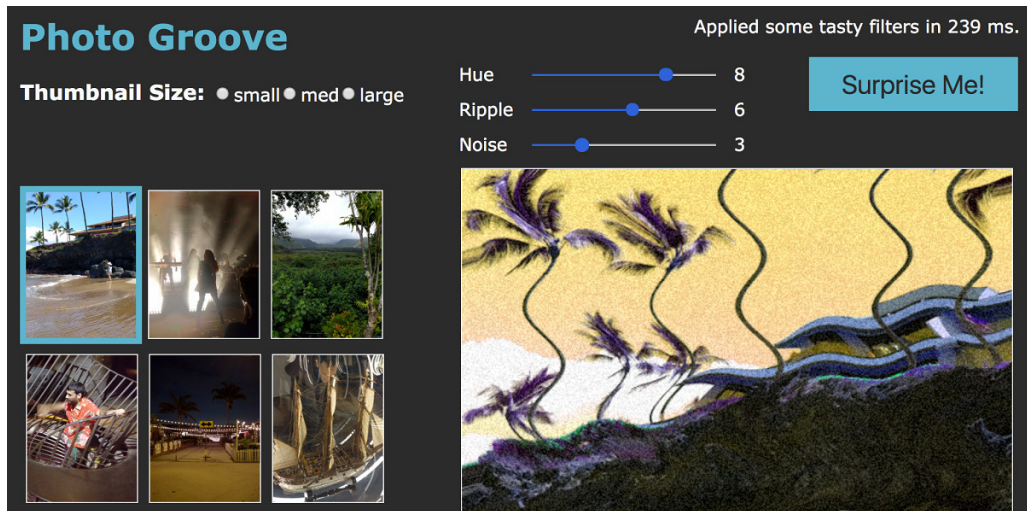


Figure 5.1 Adding sliders which control filtering for the large photo

Each slider controls an integer value which the filtering library will use for the magnitude of the filter effect. Sliding all three to 0 would disable filtering—leaving the photos unaltered, like how they look now—whereas setting all three to the maximum value of 11 would apply each filter at full power.

Implementing the sliders using Custom Elements will involve three steps:

1. Run some JavaScript code before our Elm code executes, to register custom behavior for all `<paper-slider>` elements on the page.
2. Write Elm code to render these `<paper-slider>` elements using plain `Html` values.
3. Write custom event handlers to update our model based on the sliders' states.

Along the way, we'll learn a new use for the JSON Decoders we saw in Chapter 4, and compare two alternative ways of integrating the sliders into our data model. Let's get to it!

5.1.1 Importing Custom Elements

The Paper Slider we'll be using is implemented as a Custom Element, which is a part of the Web Components specification. Here's how Custom Elements work:

- Someone writes some JavaScript which defines a new element type. In this case the

new element is a `<paper-slider>`.

- We run that JavaScript code when our page loads, to register this custom element with the browser.
- From now on, whenever any code—in JavaScript or in Elm—creates an element whose tag name is "paper-slider", that element will behave according to this custom logic.

WARNING Custom Elements are implemented in JavaScript, so they may throw runtime exceptions!

REGISTERING CUSTOM ELEMENTS

To register the custom `<paper-slider>` element on our page, we'll first need to import the code for it. Let's add a `<link rel="import">` to the `<head>` of our `index.html` file, right below the `<link rel="stylesheet">` for our stylesheet:

```
<link rel="stylesheet" href="http://elm-in-action.com/styles.css">
<link rel="import" href="http://elm-in-action.com/wc/slider/paper-slider.html">
```

This `<link>` tag loads the JavaScript code, CSS styles, and markup necessary to describe a custom `<paper-slider>` element, and then registers that `<paper-slider>` on the page. From now on, whenever an element with the tag name "paper-slider" appears on this page, it will behave according to the implementation we've imported here.

You can find more Custom Elements at <http://webcomponents.org/>, or create and serve your own from any URL you like.

Polyfilling Custom Elements for Older Browsers

Since custom elements and `<link rel="import">` tags are relatively new features, not all older browsers support them.

We can fix this for browsers as old as Internet Explorer 11 by loading some JavaScript code known as a *polyfill*, which patches support for custom elements into these browsers. To polyfill support for Custom Elements, add this `<script>` between your two `<link>` tags:

```
<link rel="stylesheet" href="http://elm-in-action.com/styles.css">
<script src="http://elm-in-action.com/polyfill/webcomponents.js"></script>
<link rel="import" href="http://elm-in-action.com/wc/slider/paper-slider.html">
```

If your users only run modern browsers which have already implemented Custom Elements, you don't need the polyfill and can omit this `<script>`. (That said, the polyfill has no effect on browsers which support Custom Elements, so it should be harmless aside from being more for users to download.)

Custom elements from the Polymer Project use something called Shady DOM to improve performance in browsers that do not natively support certain parts of the Web Components specification. However, if you use Polymer elements inside other Polymer elements, they may not work properly in their stock configuration due to how Shady DOM interacts with virtual DOM systems such as Elm's.

A workaround is to disable Shady DOM by adding this before your first `<link rel="import">` tag:

```
<script>window.Polymer = {dom: "shadow"};</script>
```

For more information on configuring Shady DOM, see <https://www.polymer-project.org/1.0/docs/devguide/settings>

In practice, polyfills are never as good as proper support for a browser feature. This polyfill often incurs significant performance penalties for some Custom Elements, and users have reported some polyfilled Custom Elements not working on Mobile Safari 10. Be sure to cross-browser test! See <http://webcomponents.org/polyfills> to learn more.

ADDING A <PAPER-SLIDER> TO THE PAGE

Back at the beginning of Chapter 2 we saw the `Html.node` function. It takes three arguments:

1. Tag name
2. List of attributes
3. List of children

Here are two ways to create the same button, one using `node` and the other using `button`:

```
node "button" [ class "large" ] [ text "Send" ]
button [ class "large" ] [ text "Send" ]
```

Functions that create elements like `button`, `label` and `input` have tiny implementations—they do nothing more than call `node`. For example, `label` and `input` can be implemented like so:

```
label attributes children =
  node "label" attributes children

input attributes children =
  node "input" attributes children
```

TIP Because of partial application, we could also have written these as `label = node "label"` and `input = node "input"` - either way works!

Now that the `<paper-slider>` custom element has been registered on the page with our `<link>` tag, the only Elm code necessary to use it is to call `node "paper-slider"`.

Let's add a function that does this, at the end of `PhotoGroove.elm`:

```
paperSlider =
  node "paper-slider"
```

This `paperSlider` function will work the same way as functions like `button`, `div`, and so on. It also has the same type as they do:

```
paperSlider : List (Attribute msg) -> List (Html msg) -> Html msg
```

Go ahead and add this type annotation right above our `paperSlider` implementation.

VIEWING THE SLIDERS

Next let's invoke `paperSlider` to render the sliders. Add this beneath the `view` function:

Listing 5.1 viewFilter

```
viewFilter : String -> Int -> Html Msg
viewFilter name magnitude =
  div [ class "filter-slider" ]
    [ label [] [ text name ]           ❶
      , paperSlider [ max "11" ] []    ❷
      , label [] [ text (toString magnitude) ] ❸
    ]
```

- ❶ Display the filter's name
- ❷ <paper-slider> that goes from 0 to 11
- ❸ Display the current magnitude

We'll also need to add `max` to the `Html.Attributes` we're exposing in our imports:

```
import Html.Attributes exposing (id, class, classList, src, name, max, type_, title)
```

If we try to compile this, we'll get a naming error: `max` is ambiguous! The compiler suggests:

Maybe you want one of the following?

```
Basics.max
Html.Attributes.max
```

The cause of this error is that Elm has a built-in math function called `max`. It's in the `Basics` module, and every Elm file has an implicit `import Basics exposing (..)` so that we can use common functions like `negate`, `not`, and `max` without having to qualify them as `Basics.negate`, `Basics.not`, or `Basics.max`.

RESOLVING AMBIGUOUS NAMES

The compiler is reporting an ambiguity in our code: when we wrote `max`, we might have meant either `Basics.max` or `Html.Attributes.max`, since both `max` functions are exposed in this file.

Let's clarify which we meant by fully qualifying `max`:

```
paperSlider [ Html.Attributes.max "11" ] []
```

This resolves the compiler error, but `Html.Attributes.max` is pretty verbose! We can shorten it by giving `Html.Attributes` an alias of `Attr` using the `as` keyword:

```
import Html.Attributes as Attr exposing (id, class, classList, src, name, type_, title)
```

Now anywhere we would write `Html.Attributes.foo` we can write `Attr.foo` instead.

Let's refactor our `input` to look like this:

```
paperSlider [ Attr.max "11" ] []
```

Splendid! This should make everything compile neatly.

RENDERING THREE SLIDERS

Now we can call `viewFilter` in our `view` function to get a few sliders rendering. Let's do that right below the code for the *Surprise Me!* button.

```
view model =
  div [ class "content" ]
    [ h1 [] [ text "Photo Groove" ]
      , button
        [ onClick SurpriseMe ]
        [ text "Surprise Me!" ]
      , div [ class "filters" ]
        [ viewFilter "Hue" 0
          , viewFilter "Ripple" 0
          , viewFilter "Noise" 0
        ]
    ]
```

Figure 5.2 shows how this looks on the page.

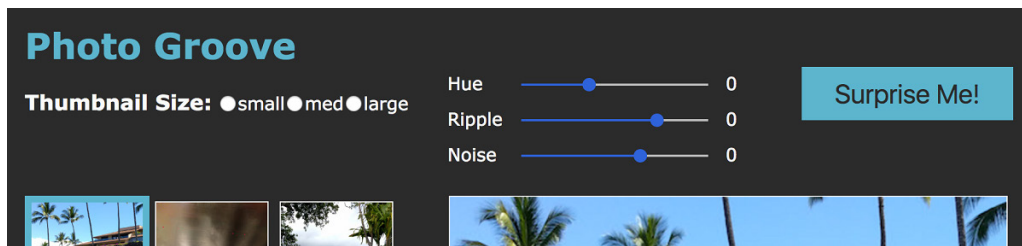


Figure 5.2 Adding `paperSliders` for Hue, Ripple, and Noise

Those sliders look slick, but they aren't completely operational yet. We want the numbers next to the sliders to change as the user slides their values around. Let's make that happen!

5.1.2 Handling Custom Events

If we take a quick glance at the Paper Slider's official documentation - hosted at <https://www.webcomponents.org/element/PolymerElements/paper-slider/paper-slider> - we can see that the slider fires a custom event, `immediate-value-changed`, as the user slides.

How might we respond to that event? `Html.Attributes` has a variety of built-in event handler functions such as `onClick`, but there's no `onImmediateValueChange`. The `immediate-value-changed` event is a custom event built into `paper-slider`, and the `Html.Attributes` module has no idea it exists. So how can we specify a handler for it?

HTML.ATTRIBUTES.ON

The `Html.Attributes.on` function lets us create a custom event handler, similarly to how the `Html.node` function lets us create a custom element. The `on` function has this type:

```
on : String -> Decoder msg -> Attribute msg
```

The `String` is the name of the event, which in this case is `"immediate-value-changed"`. The `Decoder` argument is a `Json.Decode.Decoder`, the same type of decoder we built in Chapter 4 for our HTTP responses. Here we won't be using the `Decoder` on a JSON string coming back from a server, but rather on a JavaScript event object!

DECODING IMMEDIATE-VALUE-CHANGED

The event object JavaScript uses for `"immediate-value-changed"` looks something like this:

```
{target: {immediateValue: 7}}
```

The `target` field refers to the slider object itself, which in turn has a field called `value`—an `Int` representing the slider's current value. We can use the `Json.Decode.field` and `Json.Decode.int` functions to write a decoder for this like so:

```
field "target" (field "immediateValue" int)
```

Table 5.1 compares this decoder to the example `email` decoder we wrote in Chapter 4.

Table 5.1. Decoding objects with field decoders

Decoder	JSON	Result
field "email" string	5	Err "Expected object, got 5"
	{"email": 5}	Err "Expected string for 'email', got 5"
	{"email": "cate@nolf.com"}	Ok "cate@nolf.com"
Decoder	JavaScript	Result
field "target" (field "immediateValue" int)	9	Err "Expected object, got 9"
	{"target": 9}	Err "Expected object for 'target', got 9"
	{"target": {"immediateValue": 9} }	Ok 9

JSON.DECODE.AT

There's a convenience function in `Json.Decode` for the case where we want to call `field` on another field like this: `Json.Decode.at`. It takes a list of field strings and traverses them in order. These two decoders do the same thing:

```
field "target" (field "immediateValue" int)
at [ "target", "immediateValue" ] int
```

DECODING A MSG

This `Decoder Int` will decode an integer from a JavaScript object such as `{target: {value: 7 }}`. But is that what we want? Let's look at the type of `on` again:

```
on : String -> Decoder msg -> Attribute msg
```

Notice that it wants a `Decoder msg` and then returns an `Attribute msg`. That tells us we want it to decode not an integer, but a message. Ah! We have a message type named `Msg`. So how do we convert between the `Decoder Int` that we have and the `Decoder Msg` that `on` expects?

USING JSON.DECODE.MAP

The `Json.Decode.map` function is just what the doctor ordered! It converts between decoded values, as shown in Table 5.2.

Table 5.2 `Json.Decode.map`

Expression	Description
<code>Json.Decode.map negate float</code>	Decode a float, then negate it.
<code>Json.Decode.map (\num -> num * 2) int</code>	Decode an integer, then double it.
<code>Json.Decode.map (_ -> "[[redacted]]") string</code>	Decode a string, then replace it with "[[redacted]]" no matter what it was originally. Note that this will still fail if it attempts to decode a non-string value!

Since `Json.Decode.map` takes a function which converts one decoded value to another, we can use it to convert our decoded `Int` into a `Msg`.

UPDATING IMPORTS

We'll need to expose the `on` and `at` functions in our imports in order to reference them:

```
import Html.Events exposing (onClick, on)
import Json.Decode exposing (string, int, list, Decoder, at)
```

With those changes in place, we're ready to add this `onImmediateValueChange` function to the end of `PhotoGroove.elm`:

Listing 5.2 `onImmediateValueChange`

```
onImmediateValueChange : (Int -> msg) -> Attribute msg
onImmediateValueChange toMsg =
  let
    targetImmediateValue : Decoder Int           ❶
    targetImmediateValue =
      at [ "target", "immediateValue" ] int      ❶
    msgDecoder : Decoder msg                     ❷
```



```

msgDecoder =
  Json.Decode.map toMsg targetImmediateValue
in
  on "immediate-value-changed" msgDecoder

```

- ❶ Decode the integer located at `event.target.immediateValue`
- ❷ Convert that integer to a message using `toMsg`
- ❸ Create a custom event handler using that decoder

Notice how `onImmediateValueChange` takes a `toMsg` function? This is because we need it to be flexible. We plan to have multiple sliders on the page, and we'll want each of them to have a unique `Msg` constructor so that we can tell their messages apart. The `toMsg` argument lets us pass in the appropriate constructor on a case-by-case basis, which will come in handy later.

REFACTORING TO USE PIPELINES

Notice how we assemble this value in three steps (`targetImmediateValue`, `msgDecoder`, and `on`), and each step's final argument is the previous step's return value? That means we can rewrite this to use the pipeline style we learned in Chapter 4! Let's do that refactor:

```

onImmediateValueChange : (Int -> msg) -> Attribute msg
onImmediateValueChange toMsg =
  at [ "target", "immediateValue" ] int
    |> Json.Decode.map toMsg
    |> on "immediate-value-changed"

```

Try walking through each step in the pipeline and finding the equivalent code in Listing 5.2. All the same logic is still there, just reorganized!

ADDING EVENT HANDLING TO VIEWFILTER

Now that we have `onImmediateValueChange`, we can use it in our `viewFilter` function.

Remember how we made `onImmediateValueChange` accept an `Int -> msg` function, so that we could pass it a different `Msg` constructor on a case-by-case basis? We'll want `viewFilter` to have that same flexibility. The only difference will be that since `viewFilter` returns an `Html Msg`, we won't be using a type variable like `msg`; instead, `viewFilter` will accept an `Int -> Msg` function.

Listing 5.3 using `onImmediateValueChange` in `viewFilter`

```

viewFilter : String -> (Int -> Msg) -> Int -> Html Msg
viewFilter name toMsg magnitude =
  div [ class "filter-slider" ]
    [ label [] [ text name ]
      , paperSlider [ Attr.max "11", onImmediateValueChange toMsg ] []
      , label [] [ text (toString magnitude) ]
    ]

```

- ❶ Calling `onImmediateValueChange` just like we did `onClick`

With this revised `viewFilter` implementation at the ready, our display logic is ready to be connected to our model and `update`. Once we've revised those to work with `viewFilter`, our shiny new Custom Elements will be fully integrated into our application!

5.1.3 Responding to Slider Changes

We have three filters, each of which has a name and a magnitude. How should we track their current values in our model? Let's walk through two approaches, comparing the pros and cons of each, and then at the end decide which to use.

The first approach would be to add three fields to the model: one for Hue, one for Ripple, and one for Noise. The alternative would prioritize flexibility, and store the filters as a list of { `name : String, amount : Int` } records. Table 5.3 shows these approaches side-by-side.

Table 5.3 Storing filter data as three ints versus one list of records

Three Ints	One List of Records
<pre>type alias Model = { photos : List Photo ... , hue : Int , ripple : Int , noise : Int }</pre>	<pre>type alias Model = { photos : List Photo ... , filters : List { name : String, amount : Int } }</pre>
<pre>initialModel : Model initialModel = { photos = [] ... , hue = 0 , ripple = 0 , noise = 0 }</pre>	<pre>initialModel : Model initialModel = { photos = [] ... , filters = [{ name = "Hue", amount = 0 } , { name = "Ripple", amount = 0 } , { name = "Noise", amount = 0 }] }</pre>

Each model design has its own strengths and weaknesses, which become clearer when we write code that references these parts of the model.

Let's consider how our `update` implementations might compare. In either approach, we'd expand our `Msg` type's constructors and then add at least one branch to our `update` function's *case-expression*. Table 5.4 compares how these revisions would look for each approach.

Table 5.4 Updating the model

Three Ints	One List of Records
<pre> type Msg = SelectByUrl String SetHue Int SetRipple Int SetNoise Int </pre>	<pre> type Msg = SelectByUrl String SetFilter String Int </pre>
<pre> case msg of SetHue hue -> ({ model hue = hue } , Cmd.none) SetRipple ripple -> ({ model ripple = ripple } , Cmd.none) SetNoise noise -> ({ model noise = noise } , Cmd.none) </pre>	<pre> case msg of SetFilter name amount -> let transform filter = if filter.name == name then { name = name , amount = amount } else filter filters = model.filters > List.map transform in ({ model filters = filters } , Cmd.none) </pre>

The “list of records” approach is more flexible. If we decided to add a fourth filter, such as “blur”, we could add it to `initialModel` and boom! It would appear on the page instantly. With the three integers approach, adding a fourth integer would require expanding not only `initialModel`, but also adding a field to our type alias for `Model`, a constructor for our `Msg`, and a branch for update’s *case-expression*. That sure sounds like more work!

Finally let’s compare how rendering would look.

Table 5.5 Viewing filter data from three fields versus one

Three Ints	List of Records (assuming <code>viewFilter</code> tweaked)
<pre> , div [class "filters"] [viewFilter SetHue "Hue" model.hue , viewFilter SetRipple "Ripple" model.ripple , viewFilter SetNoise "Noise" model.noise] </pre>	<pre> , div [class "filters"] (List.map viewFilter model.filters) </pre>

Assuming we tweaked `viewFilter` to accept a `{ name : String, amount : Int } record`, we could save ourselves a few lines of code with the “list of records” approach, using `List.map` as shown in Table 5.5. Once again “list of records” comes out ahead!

CHOOSING AN APPROACH

So far we’ve compared these approaches by conciseness, and also by how much effort it would take to add more filters later. However, our analysis is overlooking a crucial consideration, one which tends to make a much bigger difference in Elm than in JavaScript:

Which approach rules out more bugs?

As we’ve seen, Elm’s compiler gives us certain guarantees that can rule out entire categories of bugs. Some code can better leverage these guarantees than others! We can “help the compiler help us” by taking it into consideration when making decisions like these.

What potential future headaches would each approach let the compiler rule out?

CONSIDERING POTENTIAL BUGS

Suppose we go with the “list of records” approach and one of our coworkers makes an innocuous typo—writing “Rippl” instead of “Ripple”—such that a `SetFilter "Rippl" 5` message comes through our `update` function. That code will compile, but it won’t work properly because our `filter.name == "Rippl"` condition will never pass. We’ll have to hope our coworker catches this bug in testing!

What if our coworker makes the same typo in the three-ints approach? Attempting to create a `SetRippl 5` message will be a compiler error, because we named our `Msg` constructor `SetRipple`, not `SetRippl`.

Table 5.6 Making a Mistake

	Three Ints	List of Records
Message	<code>SetRippl 5</code>	<code>SetFilter "Rippl" 5</code>
Outcome	Elm’s compiler tells us we have a naming error.	This compiles, but now we have a bug!

CONSIDERING FUTURE CHANGES

What about making changes? Suppose in the future we need to rename “Ripple” to “Swirl”.

In the “three ints” approach, we can rename our `Msg` constructor from `SetRipple` to `SetSwirl`, and our `Model` field from `ripple` to `swirl`—and then watch Elm’s compiler tell us exactly what other parts of our code base need changing as a result. If we miss anything, we’ll get a helpful type mismatch error telling us every spot we overlooked.

With the “list of records” approach, we have to hope none of our coworkers ever wrote code using the hardcoded magic string “Ripple” instead of a constant. If they did, and

something breaks, the only way we'll find out before the bug reaches production is if someone also wrote a test that happens to fail when we change "Ripple" to "Swirl".

CHOOSING RELIABILITY

By using individual fields instead of a list of records, we can rule out the entire category of bugs related to invalid filter names.

Increasing conciseness and saving potential future effort are nice, but preventing bugs in a growing code base tends to be more valuable over time. Verbosity has a predictable impact on a project, whereas the impact of bugs can range from "quick fix" to "apocalyptic progress torpedo." Let's rule them out when we can!

We'll go with the approach that prevents more bugs. Take a moment to look back at Tables 5.3, 5.4, and 5.5, and implement the changes in the first column. Then let's revise our calls to `viewFilter` to use our model's new fields:

```
, div [ class "filters" ]
  [ viewFilter "Hue" SetHue model.hue
  , viewFilter "Ripple" SetRipple model.ripple
  , viewFilter "Noise" SetNoise model.noise
  ]
```

When the dust settles, you should be able to recompile with `elm-make --output=elm.js PhotoGroove.elm` and see the number labels change when the user slides.

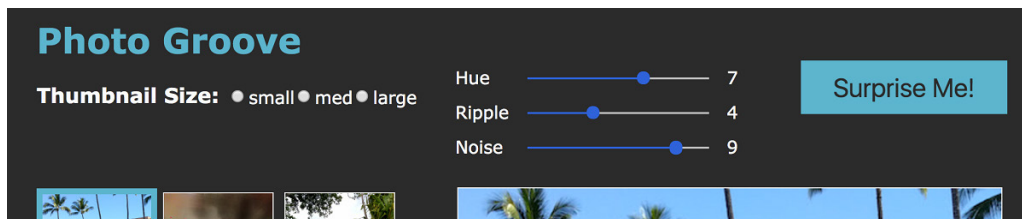


Figure 5.3 Sliding now changes the numbers next to the sliders

Now that we have our sliders set up, we can move on to introducing the filters themselves!

5.2 Sending Data to JavaScript

Now our model has all the data it needs to calibrate the filters. We'll use some JavaScript once again to apply the filters, although this time not Custom Elements. Instead we'll write code that passes configuration options and a `<canvas>` element to a JavaScript function, which will then proceed to draw groovy pictures on it.

First we'll get a basic proof-of-concept working, to confirm that we're successfully communicating across languages, and then we'll smooth out the implementation details until we're satisfied with how things are working.

5.2.1 Creating a Command using a Port

You may recall from Chapter 4 that an *effect* is an operation that modifies external state. You may also recall that if a function modifies external state when it runs, that function has a *side effect*. Elm functions are not permitted to have side effects, but JavaScript functions are.

TALKING TO JAVASCRIPT IS LIKE TALKING TO SERVERS

Since calling any JavaScript function may result in a side effect, Elm functions cannot call JavaScript functions anytime they please; this would destroy the guarantee that Elm functions have no side effects!

Instead, Elm talks to JavaScript the same way it talks to servers: by sending data out through a command, and receiving data in through a message. Figure 5.4 illustrates this.

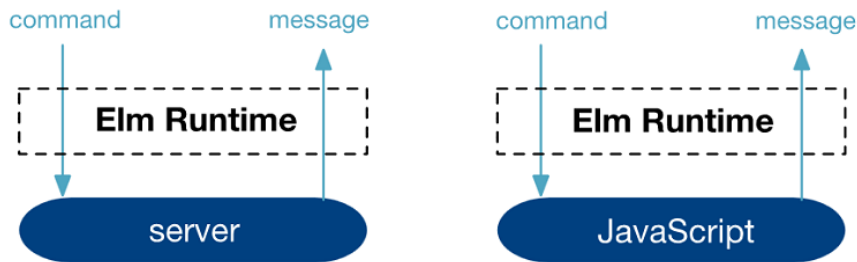


Figure 5.4 Using commands and messages to talk to both servers and JavaScript

This means that talking to JavaScript will have some characteristics in common with what we saw when talking to servers in Chapter 4:

- Data can only be sent using a command
- Data can only be received by `update`, and that data must be wrapped in a message
- We can translate and validate this incoming data using Decoders

NOTE In JavaScript, some effects are performed synchronously, with program execution halting until the effect completes. In contrast, an Elm `Cmd` always represents an *asynchronous* effect. This means that when we send data to JavaScript, it's always possible that other code might run before data gets sent back to Elm!

CREATING A COMMAND

Let's create a `Cmd` to send some data to JavaScript. First we'll define a `type alias` for the data we're going to send, right above our `type alias` for `Photo`:

```
type alias FilterOptions =
  { url : String
  , filters : List { name : String, amount : Int }
  }
```

This represents all the information our JavaScript library will need in order to apply the filters: the URL of the photo in question, plus the list of filters and their amounts.

USING A PORT TO DEFINE A FUNCTION

In Chapter 4 we created a `Cmd` using the `Http.send` function, but here we'll instead use a language feature designed specifically for talking to JavaScript: the `port` keyword.

Add this right above the `type` alias for `FilterOptions`:

```
port setFilters : FilterOptions -> Cmd msg
```

This declares a function called `setFilters`. Its type is `(FilterOptions -> Cmd msg)`. We don't write an implementation for this function, because the `port` keyword automatically writes one for us! `port` only needs to look at the type we requested to decide what the function should do.

All `port` functions that send data to JavaScript are defined using a very specific pattern:

- The `port` keyword must be followed by a function name and a type annotation
- The type annotation must be for a function that takes one argument
- The function must return `Cmd msg`, and nothing else—not even `Cmd Msg`!

TYPE VARIABLES IN VALUES

Notice that our new `setFilters` function returns `Cmd msg`, not `Cmd Msg`. We know from Chapter 3 that the lowercase “m” means `msg` is a *type variable*, like the `val` type variable we saw in `Array.fromList`:

```
fromList : List val -> Array val
```

This `List String -> Cmd msg` function is the first time we've seen a function annotation with a type variable in its return type, but no type variable in its parameters!

What does that mean? And how is `Cmd msg` different from `Cmd Msg`?

PORT COMMANDS NEVER SEND MESSAGES

Believe it or not, we've been using a `Cmd msg` for quite some time now! The official documentation for `Cmd.none` shows that it has this type:

```
none : Cmd msg
```

We'll dive deeper into values like this in future chapters, but for now it's enough to know that a `Cmd msg` is **a command which produces no message after it completes**.

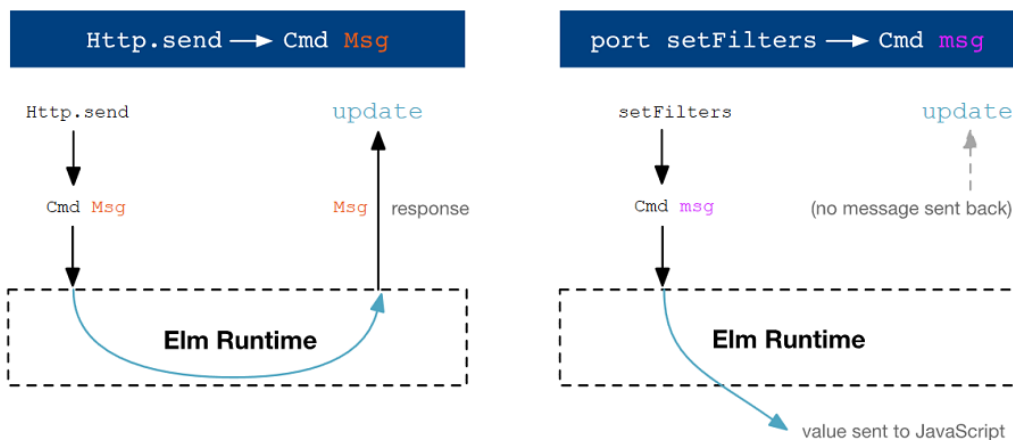
Both `Cmd.none` and `setFilters` produce no message after completing. The difference is that `Cmd.none` has no effect, whereas `setFilters` will perform the effect of sending data to JavaScript. (Specifically, it will send the `FilterOptions` value we pass it.) You can think of `setFilters` as a “fire and forget” command.

Table 5.7 Comparing `Cmd Msg` and `Cmd msg`.

Expression	Type	Effect	Message sent back to update
<pre>list photoDecoder > Http.get "http://..." > Http.send LoadPhotos</pre>	<code>Cmd Msg</code>	Send HTTP request	<code>LoadPhotos</code>
<code>Cmd.none</code>	<code>Cmd msg</code>	(none)	(no message sent back)
<code>setFilters filterOptions</code>	<code>Cmd msg</code>	Send <code>filterOptions</code> to JavaScript	(no message sent back)

NOTE While HTTP requests can fail, sending data to JavaScript cannot. We don't miss out on any error handling opportunities just because `setFilters` sends no message back to `update`.

Figure 5.5 shows the similarities and differences between a `Cmd` originating from an `Http.send` and a `Cmd` originating from a `port`.

**Figure 5.5** Comparing a `Cmd` from `Http.send` to a `Cmd` from a `port`

PORT MODULES

Any module that uses the `port` keyword must be declared using the `port module` keyword, which means we'll need to change the first line of `PhotoGroove.elm` to this:

```
port module PhotoGroove exposing (..)
```

TIP If you're ever wondering whether a given module talks directly to arbitrary JavaScript code, checking to see if it's a `port module` is usually the quickest way.

CALLING SETFILTERS WHEN USERS SELECT PHOTOS

Now that we have our port set up, we need to call `setFilters` in order to send some data to JavaScript. We'll want to apply the filters every time the user selects a photo, so that's when we'll want `update` to return the command we get from `setFilter`.

Here's one way we could modify the `SelectByUrl` branch of our `update` function to do this.

Listing 5.4 SelectByUrl

```
SelectByUrl selectedUrl ->
  let
    filters =
      [ { name = "Hue", amount = model.hue }
        , { name = "Ripple", amount = model.ripple }
        , { name = "Noise", amount = model.noise }
      ]

    url =
      urlPrefix ++ "large/" ++ selectedUrl

    cmd =
      setFilters { url = url, filters = filters }
  in
    ( model, cmd )
```

However, the `SelectByUrl` branch is not the only way a user can select a photo! Users can also click the *Surprise Me!* button to select a photo at random, and we'll want to apply filters in that situation as well. This means we'll want to reuse the above code in two places: both the `SelectByUrl` branch of our *case-expression* as well as the `SurpriseMe` branch.

SHARING CODE BETWEEN UPDATE BRANCHES

Usually the simplest way to share code is to extract common logic into a helper function and call it from both places. This is just as true for `update` as it is for any function, so let's do that!

NOTE The structure of `update` permits clever alternatives to this venerable code sharing technique—like having `update` call itself, passing a different `Msg`. This saves us from writing another function, but it's more error-prone and less performant. If we do this, and later a teammate innocently changes how that other `Msg` responds to user input, our code breaks! Having `update` call itself also runs a bit slower because we create an unnecessary `Msg` and run an unnecessary *case-expression* on it. A helper function not only runs faster, it's less error-prone to maintain because it explicitly signals to future maintainers that code is being reused.

We'll name the helper function `applyFilters`, and add it right below `update`:

Listing 5.5 applyFilters

```
applyFilters: Model -> ( Model, Cmd Msg )
applyFilters model =
  case model.selectedUrl of
    Just selectedUrl ->
```

```

    let
      filters =
        [ { name = "Hue", amount = model.hue }
        , { name = "Ripple", amount = model.ripple }
        , { name = "Noise", amount = model.noise }
        ]

      url =
        urlPrefix ++ "large/" ++ selectedUrl

    in
      ( model, setFilters { url = url, filters = filters } )

Nothing ->
  ( model, Cmd.none )

```

Now we can have both `SelectByIndex` and `SelectByUrl` call `applyFilters` directly:

```

SelectByIndex index ->
  let
    newSelectedUrl : Maybe String
    newSelectedUrl =
      ...

  in
    applyFilters { model | selectedUrl = newSelectedUrl }

SelectByUrl selectedUrl ->
  applyFilters { model | selectedUrl = Just selectedUrl }

```

Lovely! Now whenever a user clicks either a photo or the *Surprise Me!* button, `setFilters` will return a `Cmd` that sends the appropriate `FilterOptions` value over to JavaScript.

Next we'll wire up the logic on the JavaScript side, which will receive that `FilterOptions` value and use it to apply some filters!

5.2.2 Receiving Data from Elm

Now we're going to write some JavaScript code. Whenever we access a JS library from Elm, it's best to write as little JS as possible. This is because if something crashes at runtime, it's a safe bet that the culprit is somewhere in our JS code—so the less of it we have, the less code we'll have to sift through to isolate the problem.

ADDING PASTA

The JavaScript code we need comes from an open-source image filtering library intuitively named `Pasta.js`. We can import `Pasta` by adding this `<script>` tag to `index.html`, right before the `<script>` that imports our compiled `elm.js` file:

```

<div id="elm-area"></div>

<script src="http://elm-in-action.com/pasta.js"></script>
<script src="elm.js"></script>

```

This `<script>` adds a global JavaScript function called `Pasta.apply` to the page. It takes two arguments:

1. A `<canvas>` element, which is where `apply` will draw the filtered photos.
2. An `options` object, which—in a remarkable coincidence—has the same structure as the `FilterOptions` record Elm will be sending to JavaScript via our `setFilter` port.

Let's introduce that `<canvas>`. We can do this with a quick change to `PhotoGroove.elm`: having `viewLarge` render a `canvas` instead of an `img`.

Let's replace the `Just` branch of the *case-expression* in `viewLarge` with this:

```
Just url ->
img [ class "large", src (urlPrefix ++ "large/" ++ url) ] []
canvas [ id "main-canvas", class "large" ] []
```

`Pasta.apply` will take care of drawing the filtered photo onto this `canvas`. (The photo URL to draw will be sent to `Pasta.apply` via the `FilterOptions` we're sending through our port.)

RECEIVING DATA FROM THE SETFILTERS PORT

We can set up a callback function that receives data from our `setFilters` function like so:

```
<script src="http://elm-in-action.com/pasta.js"></script>
<script src="elm.js"></script>
<script>
  var app = Elm.PhotoGroove.embed(document.getElementById("elm-area"));

  app.ports.setFilters.subscribe(function(options) {
    Pasta.apply(document.getElementById("main-canvas"), options);
  });
</script>
```

We've never needed the return value of `Elm.PhotoGroove.embed` before, but now we do!

This object, typically called `app`, lets us subscribe to data Elm sends to JavaScript via ports like `setFilters`. When the Elm runtime executes the `Cmd` returned by `setFilters`, the callback function we've passed to `app.ports.setFilters.subscribe` will run. The `options` argument it accepts is the `FilterOptions` record Elm sent over, but converted from an Elm record to a JavaScript object.

WARNING Like Custom Elements, ports invoke JavaScript code—which may throw runtime exceptions. If we make a mistake in any of our JavaScript code here, Elm's compiler has no way to help us catch it.

Table 5.8 shows how a port translates immutable Elm values (like the `FilterOptions` record) into brand-new (potentially mutable) JavaScript values.

Table 5.8 Translating Elm values into JavaScript values via ports

Elm Value	Elm Type	JavaScript Value	JavaScript Type
"foo"	String	"foo"	string
4.2	Float	4.2	number
True	Bool	true	boolean
("foo", True, 4.2)	(String, Bool, Float)	["foo", true, 4.2]	Array
["drive", "line"]	List String	["drive", "line"]	Array
{ name = "Shadow" }	{ password : String }	{"name": "Shadow"}	object
Nothing	Maybe val	null	object

TIP Since all Elm values must be immutable, mutable values (like JavaScript objects and arrays) can't be sent through the port. Instead they automatically get copied into immutable data structures. This process has some overhead. Passing in values like strings and numbers, which are already immutable, has no such overhead.

Since we gave our `<canvas>` an id of "main-canvas", we can easily pass it to `Pasta.apply` by calling `document.getElementById("main-canvas")` to find it on the page by id.

TRYING IT OUT

Let's try it out! If we open the page, we see...well, an empty rectangle.

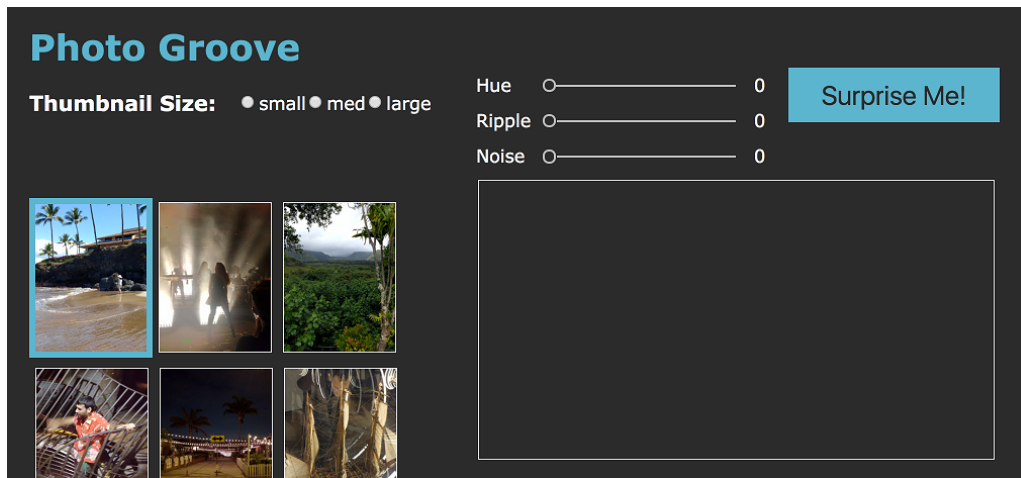


Figure 5.6 An empty rectangle. Yep.

That's not what we wanted! We'll figure that bug out later, but first let's see what these sliders do! Crank up that Ripple value, and then...hm, still nothing happens. What's the deal?

One last idea: maybe try selecting a new photo?

Oh hey, look at that!

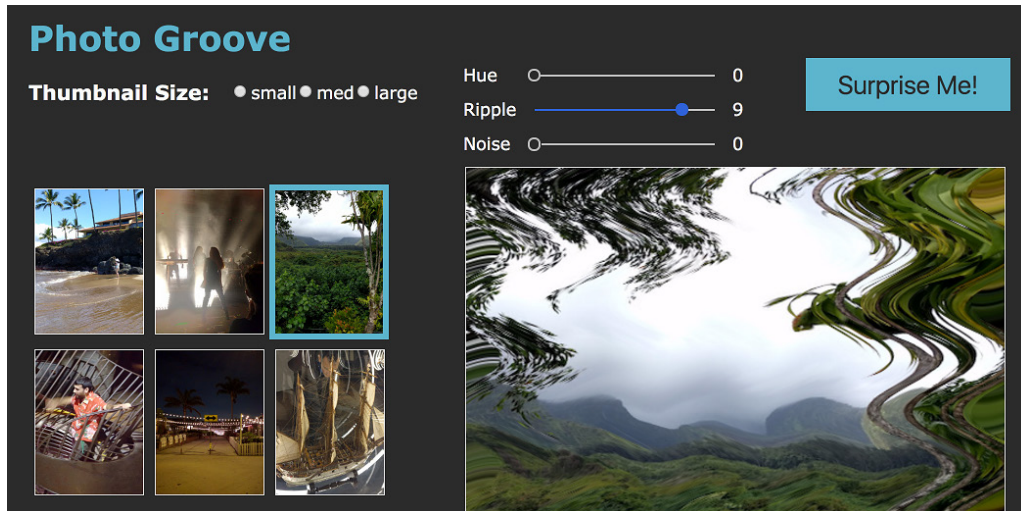


Figure 5.7 Increasing Ripple, then selecting another photo.

Sure enough, things changed...quite a lot! We are definitely talking to that JavaScript library.

However, there are some bugs to iron out. Playing around with it some more, we observe the following problems.

1. All of the Ripple and Noise values between 1 and 11 have the same effect.
2. Changing Hue doesn't seem to do anything.
3. When we initially load the page, we see an empty rectangle instead of the first photo.

Clearly we have some fixing to do!

CONVERTING THE AMOUNT VALUE TO A PERCENTAGE

The reason Hue isn't working is the same reason that the Ripple and Noise values between 1 and 11 do the same thing: we're not quite using the Pasta.js API correctly.

We're sending an `Int` between 1 and 11, but actually Pasta.js is expecting a percentage—a `Float` between 0 and 1. Since JavaScript does not draw a distinction between `Int` and `Float`, this mistake does not result in a type mismatch error—not even at runtime!—and instead, our code simply does not work as expected.

We can fix this by dividing our `Model`'s `hue`, `ripple`, and `noise` fields by 11 before sending them to JavaScript. Let's revise our logic in `applyFilters`:

```
filters =
  [ { name = "Hue", amount = toFloat model.hue / 11 }
    , { name = "Ripple", amount = toFloat model.ripple / 11 }
    , { name = "Noise", amount = toFloat model.noise / 11 }
  ]
```

We need that `toFloat` because Elm's division operator (`/`) only works if you give it two `Float` values. The `Basics.toFloat` function converts an `Int` to a `Float`, so `toFloat model.hue` converts `model.hue` from an `Int` to a `Float`—at which point we can divide it by 11 as normal.

NOTE This is another example of Elm's design emphasizing *being explicit*. If JavaScript required similar explicitness, we'd have caught this bug earlier on, when we tried to pass an `Int` to an API expecting a `Float`.

We'll also need to update `FilterOptions` to expect a `Float` for the `amount` field.

```
type alias FilterOptions =
  { url : String
    , filters : List { name : String, amount : Float }
  }
```

Now if we recompile and try our sliders again, we see a much more interesting range of Hue, Ripple, and Noise values. Lovely! There are still some bugs left, but we're making progress.

CHANGING AS WE SLIDE

This will be way more fun if the photos update every time we slide, right? We know that `applyFilters` calls our `setFilters` port, so to apply the filters every time we slide, all we need to do is to run our `applyFilters` command every time we slide:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    SetHue hue ->
      applyFilters { model | hue = hue }

    SetRipple ripple ->
      applyFilters { model | ripple = ripple }

    SetNoise noise ->
      applyFilters { model | noise = noise }
```

Recompile, and...presto! We're still greeted with an empty rectangle—we'll fix that next—but now whenever we slide, the photos update in realtime to reflect the new filter settings. Whee!

Let's fix that pesky bug where we see an empty rectangle instead of a photo on page load.

5.2.3 Timing DOM Updates

Showing the initial photo when the page loads takes two steps—a straightforward step and a tricky step. Let's start with the straightforward one: what we do after loading the photos.

APPLYING FILTERS AFTER LOADING PHOTOS

The problem here is that we currently load the photos and then re-render the view, but re-rendering the view is no longer enough to make a photo show up! We now need to call `applyFilters` so that Pasta can render something to the canvas.

To fix this, we need the `LoadPhotos` branch of our `update` function's *case-expression* to call `applyFilters` after updating `model`:

```
LoadPhotos (Ok photos) ->
  applyFilters
  { model
    | photos = photos
    , selectedUrl = Maybe.map .url (List.head photos)
  }
```

However, if you recompile and reload, you'll see the same behavior. Why wasn't that enough?

TIMING PORTS AND RENDERING

The remaining problem is a matter of timing. Let's break down what is happening:

1. We initialize `model.photos` to `[]` and `model.selectedUrl` to `Nothing`.
2. We request a list of photos from the server.
3. `view` runs, passing `Nothing` to `viewLarge` because `model.selectedUrl` is `Nothing`.
4. Because `viewLarge` received `Nothing`, it declines to render the `<canvas>`!
5. The server responds with our photos, meaning `update` gets a `LoadPhotos` message.
6. `update` runs its `LoadPhotos` branch and returns a new model (with a `selectedUrl` that is no longer `Nothing`), as well as a `Cmd` that will instruct our JavaScript code to have Pasta render to the `<canvas>`.

See the problem? Step 6 tells Pasta to render to a `<canvas>`, but as we noted in Step 4, no such `<canvas>` has been created yet!

This means our JavaScript call to `Pasta.apply(document.getElementById("main-canvas"), options)` will silently fail. Shortly after this happens, `view` will run again with the new model. This time, `model.selectedUrl` will not be `Nothing`—meaning `viewLarge` will happily ask for a fresh, blank `<canvas>`. Great.

We were so close, though! If Elm would have rendered the `view` before running the `Cmd` that invoked Pasta, the `<canvas>` would have been on the DOM before the JavaScript executed, and everything would have been fine! So why doesn't Elm do this?

OPTIMIZED DOM UPDATES

One reason the Elm Runtime has good performance is that it skips unnecessary renders.

See, browsers only repaint the DOM as pixels on users' screens every so often. If the Elm Runtime changes part of the DOM, and then changes it again before the next repaint, the first change will have been wasted time; only the second change will be painted for users to see.

You might think that if your `update` function gets called a million times in a single second, that your `view` would also be called a million times. Not so! Although those million updates will result in a million potential `Model` changes, Elm waits until the browser's next repaint to call `view` even once—with whatever `Model` is at that moment. Invoking `view` more frequently than that would result in DOM updates that the browser wouldn't bother to paint anyway.

SYNCHRONIZING WITH THE ELM RUNTIME

The JavaScript function `requestAnimationFrame` allows code to run just before the browser's next repaint. Since this is when the Elm runtime will schedule its next DOM update, we can use `requestAnimationFrame` to delay our call to `document.getElementById("main-canvas")` until after our next `view` has added the `<canvas>` we need to the DOM!

Since `requestAnimationFrame` accepts a single callback function, this means we can finally fix this bug by wrapping our call to `Pasta.apply` in a `requestAnimationFrame` callback function, like so:

```
app.ports.applyFilters.subscribe(function(options) {
  requestAnimationFrame(function() {
    Pasta.apply(document.getElementById("main-canvas"), options);
  });
});
```

TIP Anytime you need to trigger some JavaScript port code to run **after the next time** `view` results in a DOM update, you can synchronize things by wrapping your port code in `requestAnimationFrame` like this.

Let's recompile and bring up the page. Now the initial photo loads right after the page does!

5.3 Receiving Data from JavaScript

Now we're ready for the final piece of the puzzle: the status bar, which displays information reported by Pasta. First we'll start subscribing to Pasta's realtime status updates, which Pasta broadcasts as they occur. Then we'll set an initial status message which displays the version of Pasta we're using. Each change involves a different Elm concept we haven't used before: first *subscriptions*, and then *flags*. Let's put them to work!

5.3.1 Receiving realtime data from JavaScript via Ports

Here we'll be receiveing realtime data from JavaScript in the same way we receive user input: through a `Msg`. Let's start by introducing a status string to our `Model`, `initialModel`, and `Msg`.

Model	initialModel	Msg
<pre>type alias Model = { photos : List Photo , status : String }</pre>	<pre>initialModel : Model initialModel = { photos = [] , status = "" }</pre>	<pre>type Msg = SelectByUrl String SelectByIndex Int SetStatus String</pre>

With these in place, our change to `update` is straightforward: set the `status` field on `model` when we receive a `SetStatus` message.

```
update msg model =
  case msg of
    SetStatus status ->
      ( { model | status = status }, Cmd.none )
```

We'll also need `view` to display the status:

```
, button
  [ onClick SurpriseMe ]
  [ text "Surprise Me!" ]
, div [ class "status" ] [ text model.status ]
, div [ class "filters" ]
```

Nice! We're getting faster at this.

Now all that's missing is a source of `SetStatus` messages. We'll get those from JavaScript!

SENDING DATA FROM JAVASCRIPT TO ELM

Let's modify `index.html` to add a bit more JavaScript code right below our `setFilters` code.

```
var app = Elm.PhotoGroove.embed(document.getElementById("elm-area"));

app.ports.setFilters.subscribe(function(options) {
  requestAnimationFrame(function() {
    Pasta.apply(document.getElementById("main-canvas"), options);
  });
});

Pasta.addStatusListener(function(status) {
  console.log("Got a new status to send to Elm: ", status);
});
```

- ❶ Pasta calls this function whenever its status changes
- ❷ Log the new status to the developer console

This uses Pasta's `addStatusListener` function to log status changes to the browser's JavaScript console. If we reload the page and open the browser's console, we'll see various status messages flowing through as we play around with the sliders and select photos.

Next we need to send those `status` strings to Elm instead of to the console!

SUBSCRIPTIONS

Earlier in the chapter we sent data from Elm to JavaScript using a command. Now we'll do the reverse: send some data from JavaScript to Elm—using not a command, but a *subscription*.

DEFINITION A *subscription* represents a way to translate certain events outside our program into messages that get sent to our `update` function.

One use for subscriptions is handling user inputs that aren't tied to a particular DOM element. For example, we've used `onClick` to translate a click event on a specific element into a message that gets sent to `update`. What if we instead want to detect when the user resizes the entire browser window?

We do this by adding a subscription to our program, which translates global window resize events into messages. Those messages get sent to `update` just like the ones for `onClick` do.

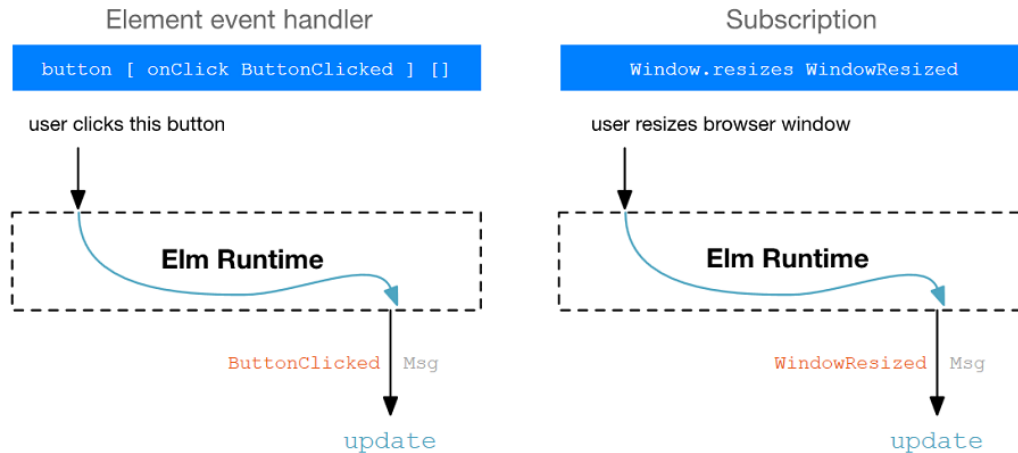


Figure 5.8 A subscription that translates browser window resize events into messages

NOTE If you'd like to try out `Window.resizes` you'll need to run `elm-package install elm-lang/window` first. You can also find subscriptions for global mouse events in the `elm-lang/mouse` package, and subscriptions for global keyboard events in the `elm-lang/keyboard` package.

DEFINING A SUBSCRIPTION PORT

There are also subscriptions which translate data from JavaScript into messages that are sent to `update`. We can get one of these subscriptions by using a slightly different `port` declaration.

Let's add another `port` right below the code that defines `setFilters`:

```
port setFilters : FilterOptions -> Cmd msg
port statusChanges : (String -> msg) -> Sub msg
```

We can already see some similarities and differences between our command port and our subscription port. Both define a function that takes one argument, but whereas the first function returned a command (`Cmd msg`), the second returns a subscription (`Sub msg`).

`Cmd` and `Sub` are both parameterized on the type of message they produce. We noted earlier how `setFilters` returns a `Cmd msg` (as opposed to `Cmd Msg`) because it is a command

which produces no message after it completes. In contrast, `statusChanges` returns a `Sub msg`, but here `msg` refers to the type of message returned by the `(String -> msg)` function we pass to `statusChanges`.

Table 5.9 shows how various calls to `statusChange` can yield different return values.

Table 5.9 Calling the `statusChanges` function

Function to pass to <code>statusChanges</code>	Expression	Return Type
<code>String.length : String -> Int</code>	<code>statusChanges String.length</code>	Sub Int
<code>String.reverse : String -> String</code>	<code>statusChanges String.reverse</code>	Sub String
<code>SetStatus : String -> Msg</code>	<code>statusChanges SetStatus</code>	Sub Msg

NOTE Whereas it's normal for `setFilters` to return `Cmd msg`, it would be bizarre for `statusChanges` to return `Sub msg`. After all, a `Cmd msg` is a command that has an effect but never sends a message to `update...` but subscriptions do not run effects. Their whole purpose is to send messages to `update`! Subscribing to a `Sub msg` would be like listening to a disconnected phone line: not terribly practical.

If we call `statusChanges SetStatus`, we'll get back a `Sub Msg` subscription. That's all well and good, but what do we do with a `Sub Msg`?

PASSING SUBSCRIPTIONS TO HTML PROGRAM

We use it with that `subscriptions` field we set up with `Html.program`!

```
main : Program Never Model Msg
main =
  Html.program
    { init = ( initialModel, initialCmd )
    , view = viewOrError
    , update = update
    , subscriptions = \_ -> Sub.none
    , subscriptions = \_ -> statusChanges SetStatus
    }
```

Ever since Chapter 3 we've been setting this `subscriptions` field to an anonymous function that always returned `Sub.none`. Now we've made that anonymous function return `statusChanges SetStatus` instead, which means that whenever JavaScript sends a string to the `statusChanges` port, it will result in a `SetStatus` message being sent to `update`.

NOTE The argument this anonymous `subscriptions` function accepts is a `Model`. Whenever our model changes, the new model is passed to this function, giving us a chance to return a different `Sub` depending on what's in the new model. This lets us dynamically control which subscriptions our program pays attention to.

Figure 5.9 illustrates how subscriptions fit into Elm programs.

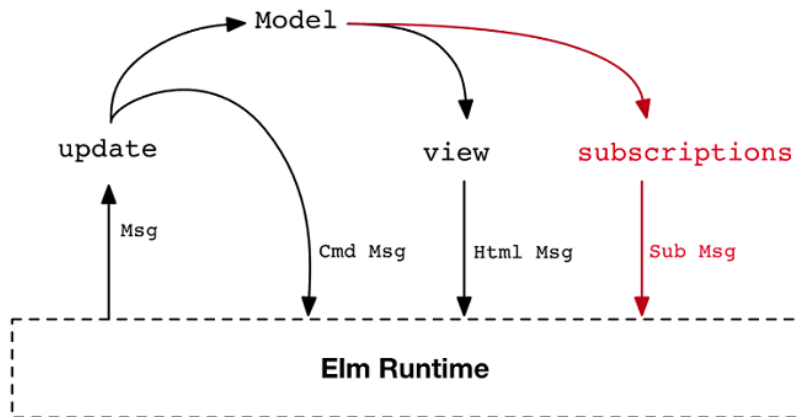


Figure 5.9 How Subscriptions fit into Elm programs

Now our `statusChanges` port is fully connected to the rest of our application, meaning we're ready to have our JavaScript code start sending it data!

CALLING `APP.PORTS.STATUSCHANGES.SEND`

Much like how we used `app.ports.setFilters.subscribe` to receive data from Elm, we can use `app.ports.statusChanges.send` to send data to Elm. Let's replace our `console.log` in `index.html` with a call to `app.ports.statusChanges.send`:

```

Pasta.addEventListener("statusChange", function(status) {
  console.log("Got a new status to send to Elm: ", status);
  app.ports.statusChanges.send(status);
});
  
```

Remember, `subscriptions = _ -> statusChanges SetStatus` specifies that we'll wrap whatever `status` string we receive from the `statusChanges` port in a `SetStatus` message.

This means that calling `app.ports.statusChanges.send("Reticulating splines")` from JavaScript will ultimately result in a `SetStatus "Reticulating splines"` message being sent to `update` on the Elm side. At that point, our existing status-rendering logic should kick in, which we can confirm by recompiling and seeing how things look.

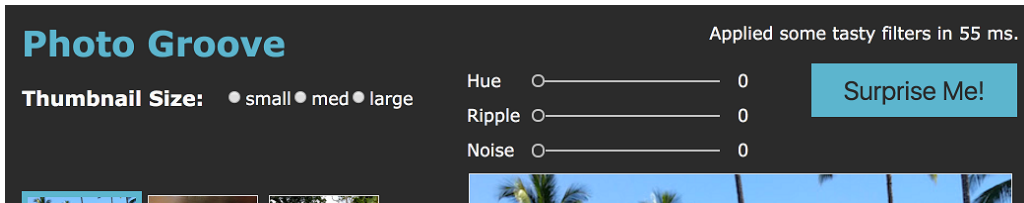


Figure 5.10 Showing the status when filters change

Fantastic! There's just one more problem to address. Right when we refresh the page, at first there is no status to show, leading to a brief flicker of nothingness before the first status update arrives. We'll fix this by showing an initial status, using data obtained from Pasta prior to the first status change it sends us.

5.3.2 Receiving initialization arguments via Flags

We'd like to display an initial status which reports the version of Pasta we're using. It will look something like this:

Initializing with Pasta v4.2...

We can easily obtain the current version from Pasta in JavaScript; all we have to do is reference `Pasta.version`. But how do we get that value into Elm in time for the initial render, to avoid that flicker of missing content?

SENDING INITIALIZATION DATA TO PROGRAMWITHFLAGS

Calling `app.ports.statusChanges.send(Pasta.version)` won't cut it here. Elm would still render `initialModel` (with `model.status` being `""`, resulting in the flicker of missing content) before receiving a message from `statusChanges` and re-rendering with the new `status`.

What we want is a way to give Elm a value from JavaScript that's available right away, early enough to use in our initial. This means we don't want ports...we want *flags*.

DEFINITION *Flags* are values passed to an Elm program's initialization function.

To use flags we'll need to switch from `Html.program` to `Html.programWithFlags`. The primary difference between the two is the type of `init` that each accepts. `program` wants `init` to be a `(Model, Cmd Msg)` tuple, whereas `programWithFlags` wants it to be a function which accepts initialization flags as an argument and returns a `(Model, Cmd Msg)` tuple.

This change lets us receive a `Float` from JavaScript—namely `Pasta.version`—as an argument to `init`, giving us the chance to incorporate it into our initial model like so:

```
main =
  Html.programWithFlags
    { init = init
    , view = viewOrError
    , update = update
    , subscriptions = (\_ -> Sub.none)
    }

init : Float -> ( Model, Cmd Msg )
init flags =
  let
    status =
      "Initializing Pasta v" ++ toString flags
  in
```

```
( { initialModel | status = status }, initialCmd )
```

Switching to `programWithFlags` affects our type annotation for `main` a bit:

```
main : Program Never Model Msg  
main : Program Float Model Msg
```

`Program Never Model Msg` means “a `Program` with `Model` as its model type, `Msg` as its message type, and which has no flags.” `Program Float Model Msg` means the same thing, except that its flags are a `Float`.

The Never Type

The type of a `Html.program` always begins with `Program Never`. What is this `Never` type, exactly?

The definition of `Never` lives in the `Basics` module, and its implementation looks something like this:

```
type Never  
  = PassMeNever Never
```

How might we obtain a value of type `Never`? That’s easy, just call `PassMeNever!` What does `PassMeNever` take as an argument? Oh, a `Never`. Hm...how can we obtain a `Never` to pass to `PassMeNever`? Ah yes—call `PassMeNever!` Now what type of argument does it take again?

To make an endlessly long story short, it’s not possible to instantiate a `Never`. Any function which accepts a `Never` for an argument is a function that can never be called! The reason `Html.program` returns a `Program Never Model Msg` is that it can never be passed any flags, `Float` or otherwise. As you may recall, `Html.program`’s `init` is not even a function, so we can never call it!

SENDING FLAGS TO ELM

On the JavaScript side, we can send flags to Elm by passing an extra argument to the `Elm.PhotoGroove.embed` method we’ve been using to start the program running.

```
var app = Elm.PhotoGroove.embed(document.getElementById("elm-area"), Pasta.version);
```

Decoding JavaScript Values

We can use the same Decoders we used in Chapter 4 to translate values from JavaScript into Elm values while gracefully handling errors.

In Chapter 4 we used the `Json.Decode.decodeString` function to decode a `String` of JSON into various Elm values. There is a similar function, `Json.Decode.decodeValue`, which takes a `Json.Decode.Value` instead of a `String`. We can write our `statusChanges` port to expect a `Value` from JavaScript, like so:

```
import Json.Decode exposing (Value)  
  
port statusChanges : (String -> msg) -> Sub msg  
port statusChanges : (Value -> msg) -> Sub msg
```

From there, writing a `Decoder Msg` and passing it to `Json.Decode.decodeValue` will give us a `Result String Msg`. We can translate that into a `Msg` by introducing an error-handling `Msg` which ideally displays a helpful message to end users, explaining that something went wrong.

We can use the same technique for `Flags`:

```
main : Program Value Model Msg
```

Using this decoder style is generally a better choice for production applications. For one, it lets us give users a better experience by gracefully handling errors. For another, any incoming type other than `Value` is liable to result in a runtime exception if JavaScript passes in something invalid!

Elm will check this immediately upon receiving the value, and will throw the exception early rather than letting it snake its way through our program, but it's better still to do the error handling ourselves. Besides, since Elm performs the automatic check using decoders anyway, it's not as if there's a performance penalty for doing it explicitly!

That's it! Now when we start up Photo Groove, we'll see the version number proudly displaying for the briefest of moments before the first photo loads. Figure 5.11 shows the initial status, captured using an incredible high-speed screenshotting camera.

Photo Groove

Initializing Pasta v4.2

Figure 5.11 Showing the status on page load

Nicely done!

5.4 Summary

Photo Groove is now substantially groovier than before. Specifically:

- We added some cool sliders using Custom Elements written in JavaScript
- As users slide their values around, it changes the filters on our large photo
- The filters come from a JavaScript library, which Photo Groove talks to like a server
- When the JS filtering library sends status reports, Photo Groove displays them
- The initial status report includes the JS filtering library's version number

In the process we learned some new concepts and techniques for building Elm applications:

- We can resolve cyclical dependency errors by making functions more flexible
- Asking "Which approach rules out more bugs?" is a good way to decide between different ways to model data
- Once Custom Elements have been registered on the page, we can access them using the `Html.node` function
- The `as` keyword lets us alias imported modules, for example `import Html.Attributes`

as Attr

- The `Html.Attributes.on` function lets us create a custom event handler using a `Decoder`
- `requestAnimationFrame` can delay our JavaScript code's execution until the next time Elm calls `view` and renders the result to the DOM
- A *subscription* represents a way to translate certain events outside our program into messages that get sent to our `update` function
- A `port` module can use the `port` keyword to define functions that return commands and subscriptions which talk to JavaScript
- `Html.programWithFlags` lets us send initialization flags from JavaScript to Elm

In Chapter 6 we'll crank our application's maintainability into overdrive by building it a hearty suite of automated tests!

Listing 5.6 index.html

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="http://elm-in-action.com/styles.css">
    <link rel="import" href="http://elm-in-action.com/wc/slider/paper-slider.html">
  </head>

  <body>
    <div id="elm-area"></div>

    <script src="http://elm-in-action.com/pasta.js"></script>
    <script src="elm.js"></script>
    <script>
      var app = Elm.PhotoGroove.embed(document.getElementById("elm-area"),
      Pasta.version);

      app.ports.setFilters.subscribe(function(options) {
        requestAnimationFrame(function() {
          Pasta.apply(document.getElementById("main-canvas"), options);
        });

        Pasta.addStatusListener(function(status) {
          app.ports.statusChanges.send(status);
        });
      });
    </script>
  </body>
</html>
```

Listing 5.7 PhotoGroove.elm

```
port module PhotoGroove exposing (..)

import Html exposing (..)
import Html.Events exposing (onClick)
import Array exposing (Array)
```



```

import Random
import Http
import Html.Attributes as Attr exposing (id, class, classList, src, name, max, type_, title)
import Json.Decode exposing (string, int, list, Decoder)
import Json.Decode.Pipeline exposing (decode, required, optional)
import Slider exposing (paperSlider)

photoDecoder : Decoder Photo
photoDecoder =
    decode buildPhoto
        |> required "url" string
        |> required "size" int
        |> optional "title" string "(untitled)"

buildPhoto : String -> Int -> String -> Photo
buildPhoto url size title =
    { url = url, size = size, title = title }

urlPrefix : String
urlPrefix =
    "http://elm-in-action.com/"

type ThumbnailSize
    = Small
    | Medium
    | Large

view : Model -> Html Msg
view model =
    div [ class "content" ]
        [ h1 [] [ text "Photo Groove" ]
        , button
            [ onClick SurpriseMe ]
            [ text "Surprise Me!" ]
        , div [ class "status" ] [ text model.status ]
        , div [ class "filters" ]
            [ viewFilter SetHue "Hue" model.hue
            , viewFilter SetRipple "Ripple" model.ripple
            , viewFilter SetNoise "Noise" model.noise
            ]
        , h3 [] [ text "Thumbnail Size:" ]
        , div [ id "choose-size" ]
            (List.map viewSizeChooser [ Small, Medium, Large ])
        , div [ id "thumbnails", class (sizeToString model.chosenSize) ]
            (List.map (viewThumbnail model.selectedUrl) model.photos)
        , viewLarge model.selectedUrl
        ]

viewFilter : (Int -> Msg) -> String -> Int -> Html Msg
viewFilter toMsg name magnitude =
    div [ class "filter-slider" ]
        [ label [] [ text name ]

```

```

    , paperSlider [ Attr.max "11", Slider.onImmediateValueChange toMsg ] []
    , label [] [ text (toString magnitude) ]
  ]

viewLarge : Maybe String -> Html Msg
viewLarge maybeUrl =
  case maybeUrl of
    Nothing ->
      text ""

    Just url ->
      canvas [ id "main-canvas", class "large" ] []

viewThumbnail : Maybe String -> Photo -> Html Msg
viewThumbnail selectedUrl thumbnail =
  img
    [ src (urlPrefix ++ thumbnail.url)
    , title (thumbnail.title ++ " [" ++ toString thumbnail.size ++ " KB]")
    , classList [ ( "selected", selectedUrl == Just thumbnail.url ) ]
    , onClick (SelectByUrl thumbnail.url)
    ]
  []

viewSizeChooser : ThumbnailSize -> Html Msg
viewSizeChooser size =
  label []
    [ input [ type_ "radio", name "size", onClick (SetSize size) ] []
    , text (sizeToString size)
    ]

sizeToString : ThumbnailSize -> String
sizeToString size =
  case size of
    Small ->
      "small"

    Medium ->
      "med"

    Large ->
      "large"

port setFilters : FilterOptions -> Cmd msg

port statusChanges : (String -> msg) -> Sub msg

type alias FilterOptions =
  { url : String
  , filters : List { name : String, amount : Float }
  }

```

```

type alias Photo =
  { url : String
  , size : Int
  , title : String
  }

type alias Model =
  { photos : List Photo
  , status : String
  , selectedUrl : Maybe String
  , loadingError : Maybe String
  , chosenSize : ThumbnailSize
  , hue : Int
  , ripple : Int
  , noise : Int
  }

initialModel : Model
initialModel =
  { photos = []
  , status = ""
  , selectedUrl = Nothing
  , loadingError = Nothing
  , chosenSize = Medium
  , hue = 0
  , ripple = 0
  , noise = 0
  }

photoArray : Array Photo
photoArray =
  Array.fromList initialModel.photos

getPhotoUrl : Int -> Maybe String
getPhotoUrl index =
  case Array.get index photoArray of
    Just photo ->
      Just photo.url

    Nothing ->
      Nothing

type Msg
  = SelectByUrl String
  | SelectByIndex Int
  | SetStatus String
  | SurpriseMe
  | SetSize ThumbnailSize
  | SetHue Int
  | SetRipple Int
  | SetNoise Int
  | LoadPhotos (Result Http.Error (List Photo))

```

```

randomPhotoPicker : Random.Generator Int
randomPhotoPicker =
    Random.int 0 (Array.length photoArray - 1)

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

        SetHue hue ->
            applyFilters { model | hue = hue }

        SetRipple ripple ->
            applyFilters { model | ripple = ripple }

        SetNoise noise ->
            applyFilters { model | noise = noise }

        SelectByIndex index ->
            let
                newSelectedUrl : Maybe String
                newSelectedUrl =
                    model.photos
                    |> Array.fromList
                    |> Array.get index
                    |> Maybe.map .url
            in
                applyFilters { model | selectedUrl = newSelectedUrl }

        SelectByUrl selectedUrl ->
            applyFilters { model | selectedUrl = Just selectedUrl }

        SurpriseMe ->
            let
                randomPhotoPicker =
                    Random.int 0 (List.length model.photos - 1)
            in
                ( model, Random.generate SelectByIndex randomPhotoPicker )

        SetSize size ->
            ( { model | chosenSize = size }, Cmd.none )

        LoadPhotos (Ok photos) ->
            applyFilters
                { model
                  | photos = photos
                    , selectedUrl = Maybe.map .url (List.head photos)
                }

        LoadPhotos (Err _) ->
            ( { model
              | loadingError = Just "Error! (Try turning it off and on again?)"
            }
            , Cmd.none

```

```

    )

applyFilters : Model -> ( Model, Cmd Msg )
applyFilters model =
    case model.selectedUrl of
        Just selectedUrl ->
            let
                filters =
                    [ { name = "Hue", amount = (toFloat model.hue) / 11 }
                    , { name = "Ripple", amount = (toFloat model.ripple) / 11 }
                    , { name = "Noise", amount = (toFloat model.noise) / 11 }
                    ]

                url =
                    urlPrefix ++ "large/" ++ selectedUrl

            in
                ( model, setFilters { url = url, filters = filters } )

        Nothing ->
            ( model, Cmd.none )

initialCmd : Cmd Msg
initialCmd =
    list photoDecoder
    |> Http.get "http://elm-in-action.com/photos/list.json"
    |> Http.send LoadPhotos

viewOnError : Model -> Html Msg
viewOnError model =
    case model.loadingError of
        Nothing ->
            view model

        Just errorMessage ->
            div [ class "error-message" ]
                [ h1 [] [ text "Photo Groove" ]
                , p [] [ text errorMessage ]
                ]

main : Program Float Model Msg
main =
    Html.programWithFlags
        { init = init
        , view = viewOnError
        , update = update
        , subscriptions = \_ -> statusChanges SetStatus
        }

init : Float -> ( Model, Cmd Msg )
init flags =
    let
        status =
            "Initializing Pasta v" ++ toString flags

```

```

in
  ( { initialModel | status = status }, initialCmd )

paperSlider : List (Attribute msg) -> List (Html msg) -> Html msg
paperSlider =
  node "paper-slider"

onImmediateValueChange : (Int -> msg) -> Attribute msg
onImmediateValueChange toMsg =
  at [ "target", "immediateValue" ] int
    |> Json.Decode.map toMsg
    |> on "immediate-value-changed"

```

6

Testing

This chapter covers

- Writing unit tests
- Writing fuzz tests
- Testing update functions
- Testing view functions

Our *Photo Groove* application has been delighting users in production for awhile now, and things have settled down on the development team. Our manager has left for vacation with vague instructions to “keep the lights on,” and we find ourselves with time to revisit some of the corners we cut when shipping *Photo Groove*.

At this point we know we’ll be maintaining this application for the long haul. We’ll need to add new features, improve existing ones, and do both without introducing bugs to the existing feature set. That isn’t easy! The longer our application is around, the more important it will be that the code we write is not just reliable, but also *maintainable*.

As we saw in Chapter 3, Elm’s compiler is an invaluable tool for maintainability. It can assist us by finding syntax errors and type mismatches before they can impact our end users, and this is even more helpful after we make a big change to our code.

However, the compiler has no way to know how our business logic is supposed to work. Fortunately, we can use another tool to verify that our business logic is still working as expected: `elm-test`. In this chapter we’ll learn how to use `elm-test` to write automated tests which verify that our business logic still works, including:

- Testing our JSON decoding logic
- Testing our `update` logic
- Testing our `view` logic

By the end, we'll have a more maintainable code base that will be easier to build on as we continue to iterate on *Photo Groove's* feature set in the chapters to come.

Let's rock!

6.1 Writing Unit Tests

Before we can write our first automated test, we'll need to introduce the `elm-test` package to our code base. Typically we'd do this using the `elm-package install` command, but setting up our tests for the first time is a bit more involved. Fortunately, the `elm-test` command line tool automates this setup for us.

6.1.1 Introducing Tests

Let's begin by running this command from the same directory as our `PhotoGroove.elm` file.

```
elm-test init
```

This will generate three things:

1. A new folder called `tests`
2. A file inside that folder called `Example.elm`
3. Another file next to `Example.elm` called `elm-package.json`

Figure 6.1 illustrates how our project's new file structure should look.



Figure 6.1 Our project's file structure after running `elm-test init`

NOTE You may also see a directory called `elm-stuff` and a file called `elm.js`. These are generated by Elm's compiler, and we can safely ignore them for the moment.

`elm-test init` also installed a package called `elm-community/elm-test`, and made its contents available only to modules in this new `tests` directory. We'll use modules from this package in the `Example.elm` file that was created inside the `tests` directory.

RENAMING EXAMPLE.ELM

Let's open up that `Example.elm` file. It should have this line at the top:

```
module Example exposing (..)
```

As vibrant a name as `Example.elm` might be, it doesn't fit very well with our application. Let's rename the file to `PhotoGrooveTests.elm` and change the first line to this:

```
module PhotoGrooveTests exposing (..)
```

NOTE Elm module names must match their filenames. If we kept the first line as `module Example` but renamed the file to `PhotoGrooveTests.elm`, we would get an error when we ran our tests.

This module will contain our *tests*, each of which is an expression that verifies something about our application's logic. Although this chapter focuses on using tests to defend against regressions in our existing application code, this is not the only way to do it! It can be helpful to write tests before finishing the initial version of the business logic itself, to guide implementation and help reveal potential design shortcomings early.

When we use the `elm-test` command to run our tests, each test will either *pass* or *fail*. Failures might indicate our application code has a problem, or that our tests have a problem—after all, tests can have bugs too! If every test passes, then the *test run* as a whole passes. If any test fails, the test run as a whole fails.

UNIT TESTS

The `suite : Test` value in our `PhotoGrooveTests` module is an example of a *unit test* in Elm. The term *unit test* means different things depending who you ask, so let's start by setting a clear definition of what it means in Elm.

DEFINITION In Elm, a *unit test* is a test which runs once, and whose test logic does not perform effects. (In Chapter 4 we learned that an effect is an operation that modifies external state.) Later in this chapter, we will encounter tests which run more than once.

We'll contrast *unit tests* with *fuzz tests* later on.

EXPECTATIONS

Every unit test requires a single expression which evaluates to an `Expectation` value.

Here's an `Expectation` that tests whether `1 + 1 == 2`.

```
expectation : Expectation
expectation =
    Expect.equal (1 + 1) 2
```

I have it on good authority that `1 + 1` should be equal to `2`, so it's safe to assume that any test which evaluates to this `Expectation` will pass when we run it.

`Expect.equal` is a function which has the type:

```
equal : a -> a -> Expectation
```

We pass it two values and it returns an `Expectation` which claims the two values are equal. If they turn out to be equal, then the expectation will pass. If not, it will fail. Failed expectations usually translate to failed tests, as we will see momentarily.

WRITING OUR FIRST UNIT TEST

Let's replace the implementation of `suite` with the following:

```
suite : Test
suite =
  test "one plus one equals two" (\_ -> Expect.equal 2 (1 + 1))
```

Figure 6.2 highlights the two arguments we've passed to the `test` function here.

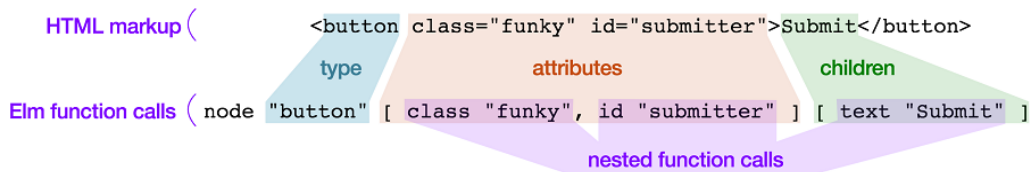


Figure 6.2 Arguments passed to the “test” function

The first argument to `test` is a description of the test. If the test fails, `elm-test` will display this text to let us know which test failed. Since the purpose of these descriptions is to help us identify which test failed, `elm-test` enforces that they must be unique. Now that we've given this test a description of "one plus one equals two", we can't give any other test that same description!

Next let's look at the argument after the description string:

```
(\_ -> Expect.equal 2 (1 + 1))
```

Oh hey! It's our friend `Expect.equal` from earlier. When the test runs, `Expect.equal` checks if the two arguments it received are equal. If they are, it returns a "pass" `Expectation`; otherwise, it returns a "fail" `Expectation`.

Since `2 == (1 + 1)`, this would be a "pass," meaning the test as a whole will pass!

Notice that what we pass to the `test` function is not an actual `Expectation` value, but rather an anonymous function which **returns** an `Expectation`. This function wrapper is important, but unit tests never need to reference the argument it receives, so we can always safely disregard that argument by naming it `"_"` the way we did here.

Why The Function Wrapper?

Although `Expect.equal 2 (1 + 1)` determines if the test passes, we don't pass that directly to the test function. We wrap it in an anonymous function first; what we actually pass is `(_ -> Expect.equal 2 (1 + 1))` instead. Why does `test` expect us to pass it a function instead of a plain old `Expectation`?

This is done in order to postpone evaluation of the test. Compare these two expressions:

```
totalA = (      10 + 20 + 30)
addNums = (\_ -> 10 + 20 + 30)
```

If we run both of these lines of code, `totalA` will immediately add `10 + 20 + 30` to get a result of 60. However, `addNums` will not do any addition yet! That line of code only defines a function. The addition won't happen until the function gets called later.

The very first release of `elm-test` did not wrap expectations in functions like this, which led to a problem on large test suites. `elm-test` would print something like "Starting test run..." followed by a blank screen, followed by a pile of output only once the last test had finished. There was no opportunity to report progress incrementally, because every test in the module began evaluating, one after the other, as soon as the module loaded!

Delaying evaluation lets `elm-test` control when and how to run each test, giving it the flexibility to do things like report incremental progress on a long-running test suite.

At this point you might be asking yourself, "Self, what exactly is the throwaway value that gets passed to that function?" Clearly the function takes an argument, because every Elm function must take at least one argument. (There is, in fact, no such thing as a zero-argument Elm function!) So what's the value that gets passed in?

To answer that, let's look at the type of the `test` function:

```
test : String -> (() -> Expectation) -> Test
```

Whoa! What's that funky `()` thing?

The `()` value is known as *Unit*. It contains no information whatsoever. It's both a value and a type; the `()` type can only be satisfied with the `()` value. We could use it to write a function like this:

```
ultimateAnswer : () -> Int
ultimateAnswer unit =
  40 + 2
```

The only way to call `ultimateAnswer` would be to pass `()` to it; since it accepts `()` as an argument, no other value but `()` will do. A function which takes only `()` as an argument returns the same exact value every time it's called, making it a very boring function. The only purpose of functions like that is to do things like delay evaluation.

In the above example, the expression `40 + 2` would not be evaluated right when the program starts; instead, it would only be evaluated when `ultimateAnswer ()` is called. This is not a big deal for a simple expression like `40 + 2`, but for a larger expression, which might take a long time to run, delaying evaluation can make a big difference!

Great! Next we'll use what we've learned to write our first unit test for *Photo Groove*.

6.1.2 Unit Testing a JSON Decoder

Take a look back at our `photoDecoder`, which we used to decode `Photo` values from the raw JSON strings our HTTP requests received from the server. `photoDecoder` describes JSON values that have two required fields and one optional field. Here's how we defined it:

```
photoDecoder : Decoder Photo
photoDecoder =
  decode buildPhoto
    |> required "url" string
    |> required "size" int
    |> optional "title" string "(untitled)"
```

We'll write a test for this code which checks if that optional field is serving the purpose we expect—namely, that if the JSON the server sends us does not contain a title field, the photo ends up with the default title of `"(untitled)"`.

First we'll write the test using only what we know now, and then we'll learn some new Elm techniques to make it nicer to read.

WRITING A FAILING TEST FIRST

Let's start by renaming `suite` to `decoderTest`, and implementing it as a failing test so we can see what failed test output looks like.

Listing 6.2 decoderTest

```
module PhotoGrooveTests exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, list, int, string)
import Test exposing (..)
import PhotoGroove exposing (..)
import Json.Decode exposing (decodeString)

decoderTest : Test
decoderTest =
  test "title defaults to (untitled)"
    (\_ ->
      "{ \"url\": \"fruits.com\", \"size\": 5 }"
        |> decodeString photoDecoder
        |> Expect.equal
          (Ok { url = "fruits.com", size = 5, title = "" })
    )
```

- ❶ Test description
- ❷ Anonymous function wrapper
- ❸ Sample JSON string
- ❹ Decoding the JSON

You may recall the `decodeString` function from Chapter 4. It takes a decoder and a string containing some JSON, then returns a `Result`.

```
decodeString : Decoder val -> String -> Result String val
```

Here we've supplied `photoDecoder` as the decoder, and `"{\"url\": \"fruits.com\", \"size\": 5}"` as the JSON string. That JSON string is pretty noisy with all those backslash-escaped quotes, though. Let's clean that up a bit.

TRIPLE-QUOTED STRINGS

Elm has a triple-quote syntax for writing longer quotes. Table 6.1 shows how we can use it to rewrite our JSON string above.

Single Quote

```
jsonSingleQuote =
    "{\"url\": \"fruits.com\", \"size\": 5}"
```

Triple Quote

```
jsonTripleQuote =
    """
    {"url": "fruits.com", "size": 5}
    """
```

The triple-quote syntax is more verbose—three quotes is, after all, more than one—but it has two advantages over the more common single-quote syntax.

1. Triple-quoted strings can span across multiple lines.
2. Triple-quoted strings can include unescaped quotation marks inside them.

Both of these are particularly useful for JSON strings. For one, they let us dodge a veritable avalanche of backslashes! For another, it's reasonable to have long JSON strings in decoder tests. In those cases, having strings which can span multiple lines really comes in handy.

Since this is a fairly short bit of JSON, we can triple-quote it but keep it on one line. Let's change `"{\"url\": \"fruits.com\", \"size\": 5}"` to `"""{"url": "fruits.com", "size": 5}"""` and keep moving.

RUNNING ALL TESTS

Our test expects that decoding this JSON string with `photoDecoder` will return `(Ok { url = "fruits.com", size = 5, title = "" })` - but in a shocking plot twist, it actually will not!

To see what it returns instead, let's re-run `elm-test` and check out the failure message.

The `elm-test` command will search our `tests` directory for `.elm` files, and run all the `Test` values they expose. In our case, `elm-test` will find `PhotoGrooveTests.elm` and an exposed `Test` value called `decoderTest`. If we added a second `.elm` file to our `tests` directory, its exposed `Test` values would also be included the next time we ran `elm-test`.

NOTE The exposing `(..)` in our module `PhotoGrooveTests` exposing `(..) declaration` means our `PhotoGrooveTests` module exposes all of its top-level values—currently just `decoderTest`—so other

modules can `import` them. Later, we'll learn how to reap some major benefits from having our modules intentionally expose less, but for now we'll continue exposing everything.

INSPECTING FAILURE OUTPUT

Let's run our tests by running this at the command line:

```
elm-test
```

We should see this output.

```

↓ PhotoGrooveTests
✗ title defaults to (untitled)

Ok { url = "fruits.com", size = 5, title = "(untitled)" }
|
| Expect.equal
|
Ok { url = "fruits.com", size = 5, title = "" }
```

TEST RUN FAILED

Figure 6.3 Output from the failed test

See the two `Ok` values with the records inside?

Those represent the arguments we passed to `Expect.equal`. They differ in that one has `title = "(untitled)"` whereas the other has `title = ""`.

It's no coincidence that the record on the bottom is the record at the end of our pipeline:

```

""{"url": "fruits.com", "size": 5}""
|> decodeString photoDecoder
|> Expect.equal
(Ok { url = "fruits.com", size = 5, title = "" })
```

As long as our test code is written in a pipeline style that ends in `|> Expect.equal`, we should be able to look at our code side-by-side with the test output and see by visual inspection which value `Expect.equal` received as which argument. The “value at the top” in our test should line up with the “value at the top” in the console output, and the “value on the bottom” in our test should line up with the “value on the bottom” in the console output.

FIXING THE FAILING TEST

The failing test output also told us why the test failed: we wrote that we expected the `title` field to be `""`, but actually it was `"(untitled)"`. (You'd think the test description we wrote of `"title defaults to (untitled)"` would have tipped us off ahead of time, but here we are.)

This test will pass once we fix the expectation. Let's change the `Expect.equal` call to this:

```
Expect.equal (Ok { url = "fruits.com", size = 5, title = "(untitled)" })
```

Now let's see if that worked, by running `elm-test` once more. The output should look like this:

TEST RUN PASSED

Figure 6.4 Output from the passing test

Huzzah! We've written our first passing test!

LEFT PIPE

Our test got easier to read when we switched to the triple-quote syntax, but we still have those clunky parentheses around our anonymous function. Is there any way we can get rid of those?

There absolutely is! In Elm, these two lines of code do the same thing:

```
String.toUpperCase (String.reverse "hello")
String.toUpperCase <| String.reverse "hello"
```

The `<|` operator takes a function and another value, and passes the value to the function. That might not sound like it does much, but it's handy for cases like the one we have here—where an infix operator would look nicer than parentheses. We can use it to refactor our test like so:

Table 6.1 Replacing Parentheses with Left Pipe

Parentheses	Left Pipe
<pre>test "title defaults to (untitled)" (_ -> ...)</pre>	<pre>test "title defaults to (untitled)" < _ -> ...</pre>

Nice! Things are looking cleaner now, but we can still improve this test.

6.1.3 Narrowing Test Scope

Our test's description says `"title defaults to (untitled)"` but we're actually testing much more than that. We're testing the structure of the entire `Photo`!

This means if we update the structure of `Photo` - say, to add a new field - this test will break. We'll have to come in and fix it manually, when we wouldn't have needed to if the test did not rely on the entire `Photo` structure. Even worse, if something breaks about `url` or `size`, we'll get failures not only for the tests that are directly responsible for those, but also for this unrelated test of `title`. Those spurious failures will clutter up our test run output!

TESTING ONLY THE TITLE

Here's how our pipeline currently works:

```
"""{"url": "fruits.com", "size": 5}"""
|> decodeString photoDecoder
|> Expect.equal (Ok { url = "fruits.com", size = 5, title = "(untitled)" })
```

Let's refactor the test to work with `title` only, by introducing a function called `Result.map`.

```
"""{"url": "fruits.com", "size": 5}"""
|> decodeString photoDecoder
|> Result.map (\photo -> photo.title)
|> Expect.equal (Ok "(untitled)")
```

We've made two changes here.

1. We added a step to our pipeline: `Result.map (\photo -> photo.title)`.
2. We changed `(Ok { url = "fruits.com", size = 5, title = "(untitled)" })` to the narrower test of `(Ok "(untitled)")`.

RESULT.MAP

We've now seen three functions called `map`: `List.map` from Chapter 2, `Maybe.map` from Chapter 4, and `Json.Decode.map` from Chapter 5. Now we're adding `Result.map` to our repertoire.

Check out the similarities between the types of `List.map`, `Maybe.map`, and `Result.map`:

```
List.map : (a -> b) -> List a -> List b
Maybe.map : (a -> b) -> Maybe a -> Maybe b
Result.map : (a -> b) -> Result x a -> Result x b
```

Pretty similar! Each `map` function...

1. Takes a data structure with a type variable we've called `a` here
2. Also takes an `(a -> b)` function that converts from `a` to `b`
3. Returns a variant of the original data structure where the type variable is `b` instead of `a`

Here are some examples of different `map` functions using `toString`. Note how `map` consistently transforms the value inside the container from an integer to a string.

```
List.map toString [ 42 ] == [ "42" ]
Maybe.map toString (Just 42) == (Just "42")
Result.map toString (Ok 42) == (Ok "42")
```

Transforming a container's contents is the main purpose of `map`, but there are also cases where `map` does not transform anything, and returns the original container unchanged.

```
List.map toString [] == []
Maybe.map toString Nothing == Nothing
Result.map toString (Err 1) == (Err 1)
```


As we can see with `Result.map`, sometimes a `map` function declines to transform an entire classification of values.

- `List.map` has no effect on `[]`
- `Maybe.map` has no effect on `Nothing`
- `Result.map` has no effect on **any** `Err` value whatsoever!

It's up to each `map` implementation to decide how to handle the values it receives, but in general they should fit this familiar pattern of transforming the contents of a container.

Inferring Implementation Details from a Type

Even without knowing their implementations, we might have been able to guess that `List.map` and `Maybe.map` would not transform empty lists or `Nothing` values. After all, if a container is empty, there's nothing to transform!

It's harder to guess that `Result.map` would ignore `Err` values, as there is no equivalent "empty container" for `Result`. Believe it or not, once you know what to look for, you can find enough information in the type of `Result.map` to be certain that it could not possibly transform `Err` values!

Here's its type again. This time let's take a closer look at the type variable that refers to `Err`, namely `x`.

```
Result.map : (a -> b) -> Result x a -> Result x b
```

Suppose the implementation of `Result.map` tried to pass an `x` value to the `(a -> b)` transformation function. That implementation wouldn't compile! `x` and `a` are different types, so passing an `x` value to a function expecting `a` would be a type mismatch. The transformation function can't possibly affect an `Err`, because it can't be passed one!

There is a function which transforms the error, called `Result.mapError`:

```
Result.mapError : (x -> y) -> Result x a -> Result y a
```

Can you spot the difference? `Result.mapError` accepts a transformation function which takes an `x` but not an `a`. It can't transform an `Ok` value because it can't be passed one, and sure enough, `Result.mapError` does nothing when passed an `Ok` value.

Taking this idea even further, have a look at the identity function, from the `Basics` module:

```
identity : a -> a
```

This function knows so little about its argument that it can't do anything to it! For example, if it tried to divide whatever it received by 2, it would have to be `identity : Float -> Float` instead. The only way to implement an Elm function with the type `a -> a` is to have it return its argument unchanged.

```
identity : a -> a
identity a = a
```

Inferring implementation details like this can speed up bug hunting. Sometimes you can tell whether a function a coworker wrote might possibly be the culprit behind a bug you're tracking down...just by looking at its type! If the function couldn't possibly affect the problematic value, then you know you're clear to move on and search elsewhere.

USING .TITLE

There's one more refactor we can make to our test. Take a closer look at the anonymous function we're passing to `Result.map`.

```
"""{"url": "fruits.com", "size": 5}"""
|> decodeString photoDecoder
|> Result.map (\photo -> photo.title)
|> Expect.equal (Ok "(untitled)")
```

In Chapter 4, we saw that `.url` is a function which takes a record and returns the contents of its `url` field. Similarly, `.title` is a function which takes a record and returns the contents of its `title` field.

Let's use `.title` in place of the equivalent `(\photo -> photo.title)` function to complete our refactor.

```
|> Result.map .title
```

THE UNIT TEST WITH NARROWED SCOPE

Here's original test we began with.

```
decoderTest : Test
decoderTest =
  test "title defaults to (untitled)"
    (\_ ->
      "{ \"url\": \"fruits.com\", \"size\": 5}"
      |> decodeString photoDecoder
      |> Expect.equal
        (Ok { url = "fruits.com", size = 5, title = "" })
    )
```

We refactored it to use triple-quoted strings, to test only the decoded title using `Result.map .title`, and to use the `<|` operator instead of parentheses around our anonymous function.

Listing 6.3 shows our refactored `decoderTest` in all its glory.

Listing 6.3 Refactored decoderTest

```
decoderTest : Test
decoderTest =
  test "title defaults to (untitled)" <|
    \_ ->
      """{"url": "fruits.com", "size": 5}"""
      |> decodeString photoDecoder
      |> Result.map .title
      |> Expect.equal (Ok "(untitled)")
```

This works great! We can now re-run `elm-test` to confirm the refactored test still passes.

However, we're only testing this one particular hardcoded JSON string. Next we'll use *fuzz tests* to expand the range of values our test covers, without writing additional tests by hand.

6.2 Writing Fuzz Tests

When writing tests for business logic, it can be time-consuming to hunt down *edge cases*—those unusual inputs which trigger bugs that never manifest with more common inputs.

In Elm, fuzz tests help us detect edge case failures by writing one test which verifies a large number of randomly-generated inputs.

DEFINITION Elm's *fuzz tests* are tests which run several times with randomly-generated inputs. Outside of Elm, this testing style is sometimes called *fuzzing*, *generative testing*, *property-based testing*, or *QuickCheck-style testing*. `elm-test` went with *fuzz* because it's concise, suggests randomness, and is fun to say.

Figure 6.5 shows what we'll be building towards.



Figure 6.5 Randomly generating inputs with fuzz tests

A common way to write a fuzz test is to start by writing a unit test and then converting it to a fuzz test to help identify edge cases.

Let's dip our toes into the world of fuzz testing by converting our existing unit test to a fuzz test. We'll do this by randomly generating our JSON instead of hardcoding it, so we can be sure our default title works properly no matter what the other fields are set to!

6.2.1 Converting Unit Tests to Fuzz Tests

Before we can switch to using randomly-generated JSON, first we need to replace our hardcoded JSON string with some code to generate that JSON programmatically.

BUILDING JSON PROGRAMMATICALLY WITH `JSON.ENCODE`

Just like how we use the `Json.Decode` module to turn JSON into Elm values, we can use the `Json.Encode` module to turn Elm values into JSON. Let's add this to the top of `TestPhotoGroove.elm`:

```
import Json.Encode as Encode
```

Since JSON encoding is the only type of encoding we'll be doing in this file, that as `Encoding` alias lets us write `Encode.foo` instead of the more verbose `Json.Encode.foo`.

JSON.ENCODE.VALUE

Whereas the `Json.Decode` module centers around the `Decoder` abstraction, the `Json.Encode` module centers around the `Value` abstraction. A `Value` (short for `Json.Encode.Value`) represents a JSON-like structure. In our case we will use it to represent actual JSON, but later we'll see how it can be used to represent objects from JavaScript as well.

We'll use three functions to build our `{"url": "fruits.com", "size": 5}` JSON on the fly:

- `Encode.int : Int -> Value`
- `Encode.string : String -> Value`
- `Encode.object : List (String, Value) -> Value`

`Encode.int` and `Encode.string` translate Elm values into their JSON equivalents. `Encode.object` takes a list of key-value pairs; each key must be a `String`, and each value must be a `Value`.

Table 6.2 shows how we can use these functions to create a `Value` representing the same JSON structure as the one our hardcoded string currently represents.

Table 6.2 Switching from String to Json.Encode.Value

<code>String</code>	<code>Json.Encode.Value</code>
<code>""{"url": "fruits.com", "size": 5}""</code>	<code>Encode.object [("url", Encode.string "fruits.com") , ("size", Encode.int 5)]</code>

JSON.DECODE.DECODEVALUE

Once we have this `Value` we want, there are two things we could do with it.

1. Call `Encode.encode` to convert the `Value` to a `String`, then use our existing `decodeString photoDecoder` call to run our decoder on that JSON string.
2. Don't bother calling `Encode.encode`, and instead swap out our `decodeString photoDecoder` call for a call to `decodeValue photoDecoder` instead.

Like `decodeString`, the `decodeValue` function also resides in the `Json.Decode` module. It decodes a `Value` directly, without having to convert to and from an intermediate string representation. That's simpler and will run faster, so we'll do it that way!

Let's start by editing our `import Json.Decode` line to expose `decodeValue` instead of `decodeString`. It should end up looking like this:

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

Then let's incorporate our new encoding and decoding logic into our test's pipeline.

Listing 6.4 Using programmatically created JSON

```
decoderTest : Test
decoderTest =
  test "title defaults to (untitled)" <|
    \_ ->
      [ ( "url", Encode.string "fruits.com" )
        , ( "size", Encode.int 5 )
      ]
      |> Encode.object
      |> decodeValue photoDecoder ❶
      |> Result.map .title
      |> Expect.equal (Ok "(untitled)")
```

❶ We now call `decodeValue` instead of `decodeString` here

FROM TEST TO FUZZ2

Now we're building our JSON programmatically, but we're still building it out of the hardcoded values `"fruits.com"` and `5`. To help our test cover more edge cases, we'll replace these hardcoded values with randomly generated ones.

The `Fuzz` module will help us do this. Thanks to `elm-test init`, it's already imported at the top of `TestPhotoGroove.elm`:

```
import Fuzz exposing (Fuzzer, list, int, string)
```

We want a randomly generated string to replace `"fruits.com"` and a randomly generated integer to replace `5`. To access those we'll make the substitution shown in Table 6.3.

Table 6.3 Replacing a Unit Test with a Fuzz Test

Unit Test	Fuzz Test
<pre>test "title defaults to (untitled)" < _ -></pre>	<pre>fuzz2 string int "title defaults to (untitled)" < \url size -></pre>

We've done two things here.

First we replaced the call to `test` with a call to `fuzz2 string int`. The call to `fuzz2` says that we want a fuzz test which randomly generates 2 values. `string` and `int` are *fuzzers* which specify that we want the first generated value to be a string, and the second to be an integer. Their types are `string : Fuzzer String` and `int : Fuzzer Int`.

DEFINITION A *fuzzer* specifies how to randomly generate values for fuzz tests.

The other change we made was to our anonymous function. It now accepts two arguments: `url` and `size`. Because we've passed this anonymous function to `fuzz2 string int, elm-test` will run this function 100 times, each time randomly generating a fresh `String` value and passing it in as `url`, and a fresh `Int` value and passing it in as `size`.

NOTE `Fuzz.string` does not generate strings completely at random. It has a higher probability of generating values that are likely to cause bugs: the empty string, very short strings, and very long strings. Similarly, `Fuzz.int` prioritizes generating 0, a mix of positive and negative numbers, and a mix of very small and very large numbers. Other fuzzers tend to be designed with similar priorities.

USING THE RANDOMLY-GENERATED VALUES

Now that we have our randomly-generated `url` and `size` values, all we have to do is to use them in place of our hardcoded `"fruits.com"` and 5 values. Here's our final fuzz test:

Listing 6.5 Our first complete fuzz test

```
decoderTest : Test
decoderTest =
  fuzz2 string int "title defaults to (untitled)" <|
    \url size -> ❶
      [ ( "url", Encode.string url )
        , ( "size", Encode.int size )
        ]
      |> Encode.object
      |> decodeValue photoDecoder
      |> Result.map .title
      |> Expect.equal (Ok "(untitled)")
```

❶ `url` and `size` come from the `string` and `int` fuzzers we passed to `fuzz2`

Great! We can now have considerably more confidence that any JSON string containing only properly-set `"url"` and `"size"` fields—but no `"title"` field—will result in a photo whose title defaults to `"(untitled)"`.

TIP For even greater confidence, we can run `elm-test --fuzz 5000` to run each fuzz test function five thousand times instead of the default of 100 times. Specifying a higher `--fuzz` value covers more inputs, but it also makes tests take longer to run. Working on a team can get us more runs without any extra effort. Consider that if each member of a five-person team runs the entire test suite ten times per day, the default `--fuzz` value of 100 gets us five thousand runs by the end of the day!

Next we'll turn our attention to a more frequently invoked function in our code base: `update`.

6.2.2 Testing update functions

All Elm programs share some useful properties which make them easier to test.

1. The entire application state is represented by a single `Model` value.

2. `Model` only ever changes when `update` receives a `Msg` and returns a new `Model`.
3. `update` is a plain old function, so we can call it from tests like any other function.

Let's take a look at the type of `update`:

```
update : Msg -> Model -> ( Model, Cmd Msg )
```

Since this one function serves as the gatekeeper for all state changes in our application, all it takes to test any change in application state is to:

1. Call `update` in a test, passing the `Msg` and `Model` of our choice
2. Examine the `Model` it returns

TESTING SELECTBYURL

Let's use this technique to test one of our simplest state changes: when a `SetUrl` message comes through the application. For reference, here's the branch of `update`'s *case-expression* that runs when it receives a `SetUrl` message.

```
SelectByUrl selectedUrl ->
  applyFilters { model | selectedUrl = Just selectedUrl }
```

This might seem like a trivial thing to test. It does so little! All it does is update the model's `selectedUrl` field, right?

Not quite! Importantly, this logic also calls `applyFilters`. What if `applyFilters` introduces a bug later? Even writing a quick test for `SelectByUrl` can give us an early warning against that and other potential regressions.

Listing 6.3 shows a basic implementation, which combines several concepts we've seen elsewhere in the chapter.

Listing 6.3 Testing SelectByUrl

```
selectByUrlSelectsPhoto : Test
selectByUrlSelectsPhoto =
  fuzz string " SelectByUrl selects the given photo by URL " <|
    \url ->
      PhotoGroove.initialModel ❶
      |> PhotoGroove.update (SelectByUrl url) ❷
      |> Tuple.first ❸
      |> .selectedUrl
      |> Expect.equal (Just url)
```

- ❶ Begin with the initial model
- ❷ Call `update` directly
- ❸ Discard the `Cmd` returned by `update`

You may recall `Tuple.first` from Chapter 1. It takes a tuple and returns the first element in it. Since `update` returns a `(Model, Cmd Msg)` tuple, calling `Tuple.first` on that value discards the `Cmd` and returns only the `Model`—which is all we care about in this case.

Let's run the test to confirm it works. Now we should see two passing tests instead of one!

TIP Notice how `elm-test` always prints "to reproduce these results, run `elm-test -fuzz 100 -seed`" and then a big number? That number is the random number seed used to generate all the fuzz values. If you encounter a fuzz test which is hard to reproduce, you can copy this command and send it to coworker. If they run it on the same set of tests, they will see the same output you will; fuzz tests are deterministic given the same seed.

Making Commands Testable

Unfortunately, `elm-test` does not currently support testing commands directly. However, you can work around this if you're willing to modify your `update` function. First, make a union type which represents all the different commands your application can run. In our case that would be:

```
type Commands
  = FetchPhotos Decoder String
  | SetFilters FilterOptions
```

Then change `update` to have this type:

```
update : Msg -> Model -> ( Model, Commands )
```

Next, write a function which converts from `Commands` to `Cmd Msg`.

```
toCmd : Commands -> Cmd Msg
toCmd commands =
  case commands of
    FetchPhotos decoder url ->
      Http.get url decoder
      |> Http.send LoadPhotos

    Setfilters options ->
      setFilters options
```

Finally, we can use these to assemble the type of `update` that `programWithFlags` expects:

```
updateForProgram : Msg -> Model -> ( Model, Cmd Msg )
updateForProgram msg model =
  let
    ( newModel, commands ) =
      update msg model
  in
    ( newModel, toCmd commands )
```

Now we can pass `updateForProgram` to `programWithFlags` and everything will work as before. The difference will be that `update` returns a value we can examine in as much depth as we like, meaning we can test it in as much depth as we like!

This technique is useful, but it is rarely used in practice. The more popular approach is to hold off on testing commands until `elm-test` supports it directly.

FUZZING COLLECTIONS

We can also use fuzzers to generate collections such as lists. For example, we can call `fuzz (list string)` to run a fuzz test which uses a `Fuzzer (List String)`.

Let's use this to test that `LoadPhotos` selects the first photo it receives.

Listing 6.3 Testing LoadPhotos

```
loadPhotosSelectsFirstPhoto : Test
loadPhotosSelectsFirstPhoto =
  fuzz (list string) "LoadPhotos selects the first photo" <|
    \urls ->
      let
        photos =
          List.map photoFromUrl urls
      in
        PhotoGroove.initialModel
          |> PhotoGroove.update (LoadPhotos (Ok photos))
          |> Tuple.first
          |> .selectedUrl
          |> Expect.equal (List.head urls)
photoFromUrl : String -> Photo
photoFromUrl url =
  { url = url, size = 0, title = "" }
```

- ❶ This `let`-expression creates a scope for our test only
- ❷ `List.head` (from Chapter 4) returns the first element in the list
- ❸ We'll use this function in future tests too

NOTE We could also have randomly generated `size` and `title` values like we did with `url` here. However, these would have been completely ignored by the test, and would make the test take longer to run. Still, it's possible that future changes to `LoadPhotos` could make it break when given different `size` and `title` values, so including them in the test would be a change worth considering despite the performance cost.

GROUPING TESTS WITH DESCRIBE

These two tests we've written are related—they both test state transitions—and it can be helpful for our test output to highlight this whenever one or both of them fails.

We can group tests together using the `describe` function, which has this type:

```
describe : String -> List Test -> Test
```

When one of the tests in the given list fails, `elm-test` will print out not only that test's description, but also the string passed to `describe` as the first argument here. For example, if we moved our `selectByUrlSelectsPhoto` test into a call to `describe "state transitions"`, its failure output would include the string `"state transitions"` like so:

```

↓ PhotoGrooveTests
↓ state transitions
X SelectByUrl selects the given photo by URL

```

Figure 6.6 Failure output after using `describe`

Let's move both of our existing test implementations inside a call to `describe`, such that they no longer have top-level type annotations.

```

stateTransitions : Test
stateTransitions =
  describe "state transitions"
    [ fuzz string "SelectByUrl selects the given photo by URL" <|
      \url ->
        ...

      , fuzz (list string) "LoadPhotos selects the first photo" <|
        \urls ->
          ...
    ]

photoFromUrl : String -> Photo
photoFromUrl = ...

```

Re-run `elm-test`. You should still see three happily passing tests!

TIP As our test suites get larger, it can be handy to run only a few tests at a time. Take a look at the `Test.skip` and `Test.only` functions in the documentation for the `elm-community/elm-test` package on the package.elm-lang.org website. We can also run only a few test files at a time by passing them as arguments to `elm-test`, for example `elm-test tests/DecoderTests.elm` or `elm-test tests/User/*.elm`.

We've now tested some decoder business logic, confirmed that running a `SelectByUrl` message through `update` selects the given photo, and verified that `LoadPhotos` selects the first photo in the list. Next we'll take the concepts we've learned so far and apply them to testing our rendering logic as well!

6.3 Testing Views

Our application has several rules concerning how we render thumbnail photos. For example:

- Initially, we don't render any thumbnails.
- Once the photos load, we render a thumbnail for each of them.
- When you click a thumbnail, that photo becomes selected.

It's important that these features keep working as we expand our application. By writing tests for each of them, we can guard against not only future business logic regressions in `update`,

but also visual regressions in `view` as well! In the remainder of the chapter, we will write tests to verify all three of these rules.

6.3.1 Testing DOM Structure

The functions to test DOM structure come from a package called `eeue56/elm-html-test`, which was also installed when we ran `elm-test init`. We can import the modules we'll be using from this package by adding this to the top of `TestPhotoGroove.elm`.

```
import Test.Html.Query as Query
import Test.Html.Selector exposing (text, tag, attribute)
```

NOTE It's common to alias `Test.Html.Query as Query` like this for brevity's sake. We'll refer to this as "the `Query` module" in this chapter, even though the fully-qualified name would be "the `Test.Html.Query` module."

With these modules in hand, we're ready to write our first `view` test!

NO PHOTOS? NO THUMBNAIIS.

The first rule we'll test is this one:

- Initially, we don't render any thumbnails.

Here's the test we'll write for that.

```
noPhotosNoThumbnails : Test
noPhotosNoThumbnails =
  test "No thumbnails render when there are no photos to render." <|
    \_ ->
      PhotoGroove.initialModel
        |> PhotoGroove.view
        |> Query.fromHtml
        |> Query.findAll [ tag "img" ]
        |> Query.count (Expect.equal 0)
```

There are several steps in this pipeline, so let's break them down into two parts.

BUILDING A QUERY.SINGLE

The first part of the pipeline is this:

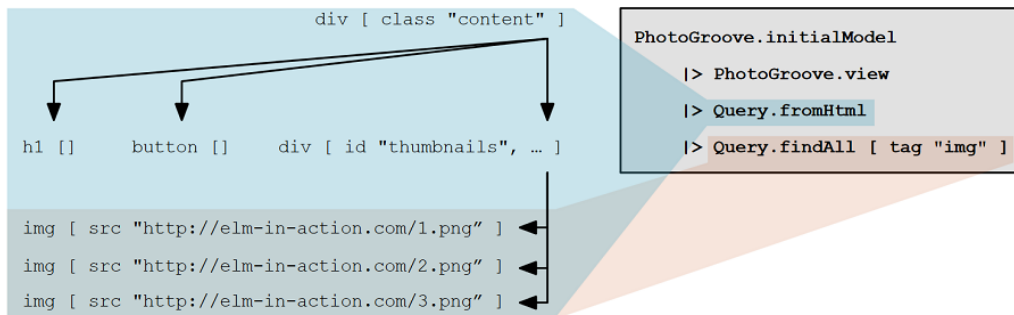
```
PhotoGroove.initialModel
  |> PhotoGroove.view
  |> Query.fromHtml
```

This code kicks things off by building a representation of the DOM that our test can examine. Table 6.5 breaks down the types involved in this expression.

Table 6.5 Types involved in `initialModel |> view |> Query.fromHtml`

Expression	Type
<code>PhotoGroove.initialModel</code>	<code>Model</code>
<code>PhotoGroove.view</code>	<code>Model -> Html Msg</code>
<code>Query.fromHtml</code>	<code>Html msg -> Query.Single msg</code>
<code>PhotoGroove.initialModel</code> <code> > PhotoGroove.view</code> <code> > Query.fromHtml</code>	<code>Query.Single msg</code>

Passing `initialModel` to `view` gives us the `Html` which will display on *Photo Groove*'s initial render. The functions in the `Query` module describe descend into this `Html`, starting from its root node and proceeding down the tree of DOM nodes it describes. Once we've narrowed it down to the nodes we care about, we call one final function to verify something about them, and produce an `Expectation`.

Figure 6.7 Finding all the `img` tags in the DOM

QUERYING HTML

`Query` functions make use of two different types as they descend into `Html`:

- `Query.Single`, which represents a single DOM node
- `Query.Multiple`, which represents multiple DOM nodes

Some functions work in terms of `Single` nodes, whereas others work in terms of `Multiple` nodes. Representing these as distinct types makes it easier to tell when we expect a query to

have one result versus several. For example, the `Query.count` function takes `Multiple`, whereas the `Query.children` function takes `Single` and returns `Multiple`.

CONVERTING FROM HTML TO QUERY

The `Query.fromHtml` function begins the process of descending into a `Html` value, by returning a `Single` representing the `Html`'s root node.

What is that root node in our case? We can tell by looking at the `view` function which returned that `Html`, namely `PhotoGroove.view`:

```
view : Model -> Html Msg
view model =
  div [ class "content" ]
```

Since `view` returns a `div` with a class of `"content"`, when our `Query.fromHtml` function returns a `Query.Single` value, that `Single` value will refer to this `div`.

NOTE The complete type of `Query.fromHtml` is `fromHtml : Html msg -> Query.Single msg`. Notice the type variable `msg` in there? Since `view` returns a `Html Msg`, calling `Query.fromHtml` will return a `Query.Single Msg` value.

CONVERTING FROM QUERY TO EXPECTATION

Now that we have our `Single` value representing the `div [class "content"]` at the root of our `Html`, let's look at the rest of the pipeline—where we transform that `Single` into an `Expectation`.

Table 6.6 breaks down the types involved in this process.

Expression	Type
<code>PhotoGroove.initialModel</code> > <code>PhotoGroove.view</code> > <code>Query.fromHtml</code>	<code>Query.Single msg</code>
<code>Query.findAll [tag "img"]</code>	<code>Query.Single msg -> Query.Multiple msg</code>
<code>Query.count (Expect.equal 0)</code>	<code>Query.Multiple msg -> Expectation</code>

We've used two functions to get from `Single` to `Expectation`: `Query.findAll` and `Query.count`.

Here's the type of `Query.findAll`:

```
findAll : List Selector -> Single msg -> Multiple msg
```

We give it a list of selectors—ways to identify a particular DOM node—and a `Single`, and it returns a `Multiple` representing all the descendant nodes which match the given selectors. We gave it only one selector—`tag "img"`—which means `findAll` will return all the `img` tags in the DOM tree beneath the `div ["content"]` at the root of our `Single` query.

Finally, once we have our `Multiple` nodes, we're ready to make a claim about them. In this case, our test is that we expect no thumbnails to be rendered, so we want to make the claim that the `Multiple` query contains zero nodes. This is where `Query.count` comes in!

```
count : (Int -> Expectation) -> Multiple msg -> Expectation
```

PARTIALLY APPLYING EXPECT.EQUAL

Notice that `Query.count` takes an `(Int -> Expectation)` function, which we satisfied by passing it a partially-applied `Expect.equal`:

```
Query.count (Expect.equal 0)
```

As you may recall from earlier, `Expect.equal` takes two arguments and compares them to verify that they're equal. That means these two expressions are equivalent:

```
Query.count (Expect.equal 0)
Query.count (\count -> Expect.equal 0 count)
```

One way to read `Query.count (Expect.equal 0)` is "Count the results in the query, and expect that count to equal 0."

COMPLETE VIEW TEST

Here's the complete implementation of our `noPhotosNoThumbnails` test, from `Model` to `Expectation` by way of `Html` and some `Query` operations.

Listing 6.7 noPhotosNoThumbnails

```
noPhotosNoThumbnails : Test
noPhotosNoThumbnails =
  test "No thumbnails render when there are no photos to render." <|
    \ ->
      PhotoGroove.initialModel
        |> PhotoGroove.view           ❶
        |> Query.fromHtml           ❷
        |> Query.findAll [ tag "img" ] ❸
        |> Query.count (Expect.equal 0) ❹
```

- ❶ Generate `Html` by rendering `initialModel`
- ❷ Return `Query.Single` for that `Html`'s root `div`
- ❸ Find the `img` nodes inside that `Query.Single`
- ❹ Count the `img` nodes, expect it to be 0

Sure enough, when we run the test, it should pass!

Next we'll expand this test to verify not only views resulting from the initial model, but also ones from models that have photos loaded.

6.3.2 Fuzzing View Tests

The second business rule we want to test is this one:

- Once the photos load, we render a thumbnail for each of them.

CHECKING IF A GIVEN THUMBNAIL WAS RENDERED

Let's start by writing a function to check whether a given URL has been rendered as a thumbnail.

```
thumbnailRendered : String -> Query.Single msg -> Expectation
thumbnailRendered url query =
  query
    |> Query.findAll [ tag "img", attribute "src" (urlPrefix ++ url) ]
    |> Query.count (Expect.atLeast 1)
```

This function starts with a `Query.Single`—which will represent the root of our page's DOM—and finds all the `img` elements within it that have the expected `src` attribute. Then runs a `Query.count` which expects that our query found at least one `img` element like that.

NOTE It's better to use `Expect.atLeast 1` than `Expect.equal 1` because our business logic permits duplicate thumbnail URLs, and we wouldn't want our test to incorrectly reject repeats as invalid!

We use `(urlPrefix ++ url)` for our `src` check instead of `url` because that's what our `viewThumbnail` implementation does. The `urlPrefix` is `"http://elm-in-action.com/"`, so without prepending that, we would be searching for a `src` of `"foo.jpeg"` when in actuality a `src` of `"http://elm-in-action.com/foo.jpeg"` was rendered—meaning this test would never pass.

CHECKING IF ALL THUMBNAILS WERE RENDERED

Now that we have a function to test whether a single thumbnail was rendered, let's use it to build a function which tests whether *all* thumbnails were rendered. We could test this by hardcoding some example photos, but why bother? We can get better test coverage by using fuzz tests to randomly generate URLs, and build photos from those!

We want to generate a list of URLs, so we might think to start by using a fuzz test which generated a list of strings like we did with our `LoadPhotos` test earlier:

```
fuzz (list string) "URLs render as thumbnails" <|
```

However, there's a subtle problem here: the list fuzzer can potentially generate hundreds of elements. That might not sound like a big deal, but let's recap what we want this test to do.

1. Generate a list of URLs.

2. Render a page which creates a `thumbnail Html` element for each of these URLs
3. Convert that entire `Html` structure to a `Query.Single`
4. Traverse every element inside the `Query.Single` looking for an `img` with the right `src`
5. Repeat the previous step for every single one of the URLs

If the list fuzzer generates 300 URLs on average, that means for each of those 300 URLs, `Query.findAll` has to traverse 300 nodes searching for the one that matches. That's a total of 90,000 nodes to traverse, and we're not even done yet! Remember how fuzz tests run 100 times by default? If those tests generate 300 URLs on average, the total traversal count would be 300 times 300 times 100—so we'd be looking at **9 million traversals** in a single test run.

That test would take, uh...a bit of time to run.

So let's not do that! When test suites start taking a long time to run, it slows down the whole team. Tests accumulate, and it's important to be mindful of their performance characteristics to avoid letting slowdowns accumulate as well.

CUSTOMIZING LIST SIZES

We can prevent this combinatoric explosion by restricting the sizes of the lists we generate.

Let's start our test out like this:

```
thumbnailsWork : Test
thumbnailsWork =
  fuzz (Fuzz.intRange 1 5) "URLs render as thumbnails" <|
    \urlCount ->
```

`Fuzz.intRange` will randomly generate an integer between 1 and 5. We'll use it to build a list between 1 and 5 elements. (We already have a test covering the zero-elements case: our `noPhotosNoThumbnails` test from earlier.)

Here's how we can convert from `urlCount` to a list of URLs.

```
urls : List String
urls =
  List.range 1 urlCount
    |> List.map (\num -> toString num ++ ".png")
```

The `List.range` function takes two integers and makes a list containing all the numbers in between. We can see it in action in `elm-repl`:

```
> List.range 1 5
[1,2,3,4,5] : List Int
```

Now that we have a reasonably-sized list of URLs, we can build photos from them using the `photoFromUrl` function we wrote earlier. Then we can build a query using a pipeline similar to the one created for our last `view` test. This time, instead of leaving `initialModel` unchanged, we'll override its `photos` field to be the list of photos we just created using `photoFromUrl` on the fly.

```
{ initialModel | photos = List.map photoFromUrl urls }
|> PhotoGroove.view
```



```
|> Query.fromHtml
```

This gives us a `Query.Single`, which we can use to create an `Expectation` to complete our test.

COMBINING EXPECTATIONS WITH EXPECT.ALL

Let's pause and take a look at what we've put together so far. We have:

1. A `Query.Single` representing the root DOM node of our page
2. A function which checks if a `Query.Single` renders the given URL as a thumbnail:

```
thumbnailRendered : String -> Query.Single msg -> Expectation
```
3. A list of URLs: `urls : List String`

We can distill these down to a single `Expectation` using the `Expect.all` function:

```
Expect.all : List (subject -> Expectation) -> subject -> Expectation
```

The purpose of `Expect.all` is to run a series of checks on a single subject value, and return a passing `Expectation` only if they all pass. In our case, the subject will be the `Query.Single` value we've built up, meaning we need to pass `Expect.all` a list with this type:

```
List (Query.Single msg -> Expectation)
```

We can use `thumbnailRendered` to get just such a list! Let's take a look at its type again:

```
thumbnailRendered : String -> Query.Single msg -> Expectation
```

We can get our list of checks by partially applying `thumbnailRendered` like so:

```
thumbnailChecks : List (Query.Single msg -> Expectation)
thumbnailChecks =
    List.map thumbnailRendered urls
```

Because we've passed a `List String` to `List.map`, it will pass each `String` in that list as the first argument to `thumbnailRendered`. The result of this `List.map` is the `List (Query.Single msg -> Expectation)` we needed for `Expect.all`.

THE FINAL TEST

Putting it all together, here's what we get.

Listing 6.7 The complete thumbnail test

```
thumbnailsWork : Test
thumbnailsWork =
    fuzz (Fuzz.intRange 1 5) "URLs render as thumbnails" <|
        \urlCount ->
            let
                urls : List String
                urls =
                    List.range 1 urlCount
                    |> List.map (\num -> toString num ++ ".png")
```

```

        thumbnailChecks : List (Query.Single msg -> Expectation)
        thumbnailChecks =
            List.map thumbnailRendered urls
    in
        { initialModel | photos = List.map photoFromUrl urls }
        |> PhotoGroove.view
        |> Query.fromHtml
        |> Expect.all thumbnailChecks

```

Now that we've verified that the thumbnails display as expected, we can move on to testing that they work properly when a user clicks one!

6.3.3 Testing User Interactions

Our final piece of business logic we'll be testing is this:

- When you click a thumbnail, that photo becomes selected.

Here we can benefit from the test coverage we've obtained through tests we wrote earlier in the chapter. We already have tests which verify:

- When the photos load, the first one gets selected.
- Whenever update receives a `SelectByUrl` message, that photo becomes selected.

Having that logic independently verified simplifies what it takes to test this requirement: all we need to do is test that when the user clicks a thumbnail, the Elm Runtime will send an appropriate `SelectByUrl` message to `update`.

FUZZ.MAP

Once again we'll want a list of URLs, and once again we'll want to avoid a combinatoric explosion by limiting the number of photos we generate to only a handful. Now that we have multiple tests that want a short list of URL strings, we can avoid code duplication by creating a custom Fuzzer (`List String`) value that has these characteristics, using `Fuzz.map`.

Earlier in the chapter we learned about `Result.map`, and saw how it compared to `List.map`, `Maybe.map`, and `Json.Decode.map`. Buckle up, because `Fuzz.map` is going to be *super duper different* from those other `map` functions.

```

Fuzz.map : (a -> b) -> Fuzzer a -> Fuzzer b
Json.Decode.map : (a -> b) -> Decoder a -> Decoder b
List.map : (a -> b) -> List a -> List b
Maybe.map : (a -> b) -> Maybe a -> Maybe b

```

Okay, not really. `Fuzz.map` works the same way as these other `map` functions do: it takes a function which converts from one contained value to another.

Listing 6.8 shows how we can use `Fuzz.map` to implement our custom fuzzer.

Listing 6.8 Custom Fuzzer with Fuzz.map

```
urlFuzzer : Fuzzer (List String)
urlFuzzer =
    Fuzz.intRange 1 5
    |> Fuzz.map urlsFromCount

urlsFromCount : Int -> List String
urlsFromCount urlCount =
    List.range 1 urlCount
    |> List.map (\num -> toString num ++ ".png")
```

Now that we have this fuzzer, we can refactor our `thumbnailsWork` test to use it. First we'll remove the `urls` definition from inside its *let-expression*, and then we'll replace the test definition with this:

```
thumbnailsWork =
    fuzz urlFuzzer "URLs render as thumbnails" <|
    \urls ->
```

SIMULATING CLICK EVENTS

Next we'll write a fuzz test that renders a list of photos and simulates clicking one of them.

Let's think about the structure of that test. It needs two ingredients:

- A list of one or more photos
- Knowing which photo in that list to click

One way we could set this test up is the same way we did with `thumbnailsWork` above:

```
fuzz urlFuzzer "clicking a thumbnail selects it" <|
    \urls->
```

There's a problem with this approach. What if `urlFuzzer` happens to generate an empty list? We'd have no photo to click!

GENERATING A NONEMPTY LIST

We can fix this by generating a single URL, plus a list of other URLs to go after it. It's okay if that second list happens to be empty, because we'll still have the single URL to work with.

```
fuzz2 string urlFuzzer "clicking a thumbnail selects it" <|
    \urlToSelect otherUrls ->
    let
        photos =
            [ urlToSelect ] ++ otherUrls
```

NOTE Another way represent this would be to use a `Nonempty List` data structure. Check out the `mgold/elm-nonempty-list` package if you're interested in learning more about `Nonempty Lists`.

THE :: OPERATOR

Incidentally, when we have a single element to add to the front of a list like this, the `::` operator is both more concise and more efficient than what we've done here.

The following two expressions produce the same list, but the version that uses `::` is more efficient because it does not create an extra list in the process.

```
[ urlToSelect ] ++ otherUrls
urlToSelect :: otherUrls
```

CHOOSING WHICH PHOTO TO CLICK

This test is looking better, but there's another problem. How do we decide which photo to click? We could decide to choose the first one every time, but then our test wouldn't be very complete. We'd only be testing that clicking the *first* thumbnail works, not that clicking *any* thumbnail works.

Could we select which photo to click at random? Well, as we saw in Chapter 3, generating random numbers is an effect (represented in Elm as a `Cmd`), and `elm-test` tests are not permitted to run effects.

What we can do instead is to generate two lists at random: one list to go before the URL we'll click, and another to go after it.

```
fuzz3 urlFuzzer string urlFuzzer "clicking a thumbnail selects it" <|
  \urlsBefore urlToClick urlsAfter ->
    let
      photos =
        urlsBefore ++ urlToClick :: urlsAfter
```

NOTE Another way to represent this would be to use a List Zipper data structure. Check out the [rtfeldman/selectlist](https://github.com/rtfeldman/selectlist) package if you're interested in learning more about List Zippers.

We've almost got the list we want, but not quite. Since we want to make sure we click the right one, its URL needs to be unique within the list of photos. We can do this by giving it a different extension from the others; we used `".png"` in our `urlFuzzer` definition, so we'll use `".jpeg"` here.

```
url =
  urlToSelect ++ ".jpeg"

photos =
  urlsBefore ++ url :: urlsAfter
```

HTML.TEST.EVENTS

It's time to simulate the click event! We'll use the `Html.Test.Event` module to do this, so let's add it at the top of `PhotoGrooveTests` and alias it to `Event` for brevity.

```
import Test.Html.Event as Event
```

This gives us access to a function called `Event.simulate` which simulates user events, such as clicks, and checks whether they result in the expected message being sent to `update`.

Here's how we'll use it.

Listing 6.10 Event.simulate

```
clickThumbnail : Test
clickThumbnail =
  fuzz3 urlFuzzer string urlFuzzer "clicking a thumbnail selects it" <|
    \urlsBefore urlToSelect urlsAfter ->
      let
        url =
          urlToSelect ++ ".jpeg"

        photos =
          (urlsBefore ++ url :: urlsAfter)
            |> List.map photoFromUrl

        srcToClick =
          urlPrefix ++ url
      in
        { initialModel | photos = photos }
          |> PhotoGroove.view
          |> Query.fromHtml
          |> Query.find [ tag "img", attribute "src" srcToClick ]
          |> Event.simulate Event.click ①
          |> Event.expect (SelectByUrl url) ②
```

- ① Simulate that a user clicked this img
- ② Expect that this message was sent to update

Those last three lines are new! Here's what's different about them.

1. We used `Query.find` instead of `Query.findAll`. This is different in that `Query.find` expects to find exactly 1 node, and returns a `Single` instead of a `Multiple`. If it finds zero nodes, or multiple nodes, it immediately fails the entire test.
2. `Event.simulate Event.click` simulates a click event on the node returned by `Query.find`. It returns an `Event` value, representing what the Elm runtime will produce in response to that `click` event.
3. `Event.expect (SelectByUrl url)` converts the `Event` into an `Expectation`, by checking that the event resulted in a `(SelectByUrl url)` message being sent to `update`.

If we run all this, sure enough, it works!

FINAL TESTS

With this test in place, we have now covered all three cases of our rendering logic that we set out to cover:

- *Initially, we don't render any thumbnails.*

- *Once the photos load, we render a thumbnail for each of them.*
- *When you click a thumbnail, that photo becomes selected.*

Great success!

6.4 Summary

We've learned plenty in this chapter, including:

- How to introduce tests to an existing Elm application
- The similarities between `Result.map`, `List.map`, `Maybe.map`, and `Json.Decode.map`
- Building JSON values using `Json.Encode`
- How to use `Json.Decode.decodeValue` instead of `decodeString`
- Describing expectations using `Expect.equal`, `Expect.atLeast`, and `Expect.all`
- Writing unit tests using the `test` function
- Writing fuzz tests by passing fuzzers to `fuzz`, `fuzz2`, and `fuzz3`
- Testing the update function by passing handcrafted messages to it in tests
- Assembling `Html` values and querying them with `Test.Html.Query`
- Descending through the DOM using `Query.findAll`
- Simulating sending events to rendered `Html` using `Event.simulate`

Here's the final `PhotoGrooveTests.elm` file.

Listing 6.11 PhotoGrooveTests.elm

```
module PhotoGrooveTests exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, list, int, string)
import Test exposing (..)
import PhotoGroove exposing (..)
import Json.Decode exposing (decodeValue)
import Json.Encode as Encode
import Test.Html.Query as Query
import Test.Html.Selector exposing (text, tag, attribute)
import Test.Html.Event as Event

decoderTest : Test
decoderTest =
    fuzz2 string int "title defaults to (untitled)" <|
        \url size ->
            [ ( "url", Encode.string url )
            , ( "size", Encode.int size )
            ]
            |> Encode.object
            |> decodeValue photoDecoder
            |> Result.map .title
            |> Expect.equal (Ok "(untitled)")

stateTransitions : Test
```

```

stateTransitions =
  describe "state transitions"
    [ fuzz string "SelectByUrl selects the given photo by URL" <|
      \url ->
        PhotoGroove.initialModel
          |> PhotoGroove.update (SelectByUrl url)
          |> Tuple.first
          |> .selectedUrl
          |> Expect.equal (Just url)
      , fuzz (list string) "LoadPhotos selects the first photo" <|
        \urls ->
          let
            photos =
              List.map photoFromUrl urls
          in
            PhotoGroove.initialModel
              |> PhotoGroove.update (LoadPhotos (Ok photos))
              |> Tuple.first
              |> .selectedUrl
              |> Expect.equal (List.head urls)
    ]

photoFromUrl : String -> Photo
photoFromUrl url =
  { url = url, size = 0, title = "" }

noPhotosNoThumbnails : Test
noPhotosNoThumbnails =
  test "No thumbnails render when there are no photos to render." <|
    \_ ->
      PhotoGroove.initialModel
        |> PhotoGroove.view
        |> Query.fromHtml
        |> Query.findAll [ tag "img" ]
        |> Query.count (Expect.equal 0)

thumbnailRendered : String -> Query.Single msg -> Expectation
thumbnailRendered url query =
  query
    |> Query.findAll [ tag "img", attribute "src" (urlPrefix ++ url) ]
    |> Query.count (Expect.atLeast 1)

thumbnailsWork : Test
thumbnailsWork =
  fuzz urlFuzzer "URLs render as thumbnails" <|
    \urls ->
      let
        thumbnailChecks : List (Query.Single msg -> Expectation)
        thumbnailChecks =
          List.map thumbnailRendered urls
      in
        { initialModel | photos = List.map photoFromUrl urls }
          |> PhotoGroove.view
          |> Query.fromHtml

```

```

|> Expect.all thumbnailChecks

urlFuzzer : Fuzzer (List String)
urlFuzzer =
  Fuzz.intRange 1 5
  |> Fuzz.map urlsFromCount

urlsFromCount : Int -> List String
urlsFromCount urlCount =
  List.range 1 urlCount
  |> List.map (\num -> toString num ++ ".png")

clickThumbnail : Test
clickThumbnail =
  fuzz3 urlFuzzer string urlFuzzer "clicking a thumbnail selects it" <|
    \urlsBefore urlToSelect urlsAfter ->
      let
        url =
          urlToSelect ++ ".jpeg"

        photos =
          (urlsBefore ++ url :: urlsAfter)
          |> List.map photoFromUrl

        srcToClick =
          urlPrefix ++ url
      in
        { initialModel | photos = photos }
        |> PhotoGroove.view
        |> Query.fromHtml
        |> Query.find [ tag "img", attribute "src" srcToClick ]
        |> Event.simulate Event.click
        |> Event.expect (SelectByUrl url)

```


7

Data Modeling

This chapter covers

- Storing values with associated keys using dictionaries
- Building interactive trees using recursive union types
- Using intermediate representations to decode JSON incrementally
- Decoding recursive data structures with recursive JSON decoders

Our manager is back from vacation, and is eager to tell the team about a poolside revelation for a new *Photo Groove* feature: *Photo Folders*. (The original name was “Photo Pholders” but Marketing said *no*.) As our users’ photo collections grow, they’ve been asking for ways to organize them. Folders are a tried-and-true metaphor for this!

To build this feature, we’ll introduce a second page to *Photo Groove*. It will let users navigate through a folder hierarchy, with each folder potentially holding several photos as well as several other folders.

This Photo Folders page will also showcase a new feature one of our coworkers built: Related Photos. This feature automatically analyzes a user’s photos, detects which ones are related to which others, and tags those photos as related on the server. The Photo Folders page will be the first to display these new Related Photo relationships! When the user selects a photo in a particular folder, we’ll show them a larger version of the photo, as well as thumbnails of its related photos.

As we implement all this, we’ll introduce some new concepts we haven’t seen before. To implement the Selected Photo display we’ll try out a new data structure: a *dictionary*. Once we’re done with that, we’ll see how to build our Folders display using a *recursive union type*, and how to model its expanding and collapsing using *recursive messages*. Finally we’ll write some JSON decoders to load all this from the server, using *intermediate representations* and *recursive decoders* to handle the nested JSON format our server will send us for this page.

Let's get started!

7.1 Storing values by keys in Dictionaries

The new Photo Folders page we'll be building will have two sections: the folders themselves, followed by a large display of the Selected Photo and its Related Photos. When the user clicks a photo in either the Folder display or in the Related photos display, that photo will become the Selected Photo that we show in the Selected Photo display, along with its related photos.

Figure 7.1 shows a screenshot of the page we'll be building up to in this chapter.



Figure 7.1 The Photo Folders page we'll be building up to in this chapter

Since we can't implement the Folder display's "click to select a photo" functionality until the Selected Photo display exists, we'll implement the Selected Photo display first and build the Folder display afterwards.

7.1.1 Setting up the Page

It's normal to put the code for each page in an Elm application in its own `.elm` file. This means we'll be giving the `PhotoGroove.elm` file we've been working on in the previous chapters a sibling `.elm` file!

Our new file will have its own `Model`, `view`, `update`, and `main`. Like `PhotoGroove.elm`, the first thing it will do when the page loads is to fetch `Model` data from the server using the `Http` module, similarly to how we did it in Chapter 4. The new file will be completely independent of `PhotoGroove.elm`—at least until we start linking the two pages together in Chapter 8!

Over the last several chapters we've built up `PhotoGroove.elm` gradually. First we used a record for our `Msg`, and then later a union type. We also moved from `Html.beginnerProgram`

to `Html.program`. This time we can use the knowledge we've acquired over the previous chapters to cut right to the chase, by using `Html.program` with a union type for our `Msg` from the get-go!

Let's create a new file named `PhotoFolders.elm`, and give it these contents:

Listing 7.1 PhotoFolders.elm

```

module PhotoFolders exposing (main) ❶

import Http
import Json.Decode as Decode exposing (Decoder, int, list, string) ❷
import Json.Decode.Pipeline exposing (required)

type alias Model =
  { selectedPhotoUrl : Maybe String ❸
  }

initialModel : Model
initialModel =
  { selectedPhotoUrl = Nothing }

init : ( Model, Cmd Msg )
init =
  ( initialModel
  , modelDecoder
    |> Http.get "http://elm-in-action.com/folders/list"
    |> Http.send LoadPage
  ) ❹ ❺ ❻ ❼

modelDecoder : Decoder Model
modelDecoder =
  Decode.succeed initialModel ❽

type Msg ❷
  = SelectPhotoUrl String ❷
  | LoadPage (Result Http.Error Model) ❷

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    SelectPhotoUrl url -> ❸
      ( { model | selectedPhotoUrl = Just url }, Cmd.none ) ❸
    LoadPage (Ok newModel) -> ❹
      ( newModel, Cmd.none ) ❹
    LoadPage (Err _) -> ❺
      ( model, Cmd.none ) ❺

```

❶ We'll define main later

- ② ``as Decode`` lets us call (for example) `Decode.succeed` instead of `Json.Decode.succeed`
- ③ The selected photo, if the user has selected one
- ④ Use `initialModel` as the model when the page loads
- ⑤ Begin a HTTP request to load data from the server when the page loads
- ⑥ For now, ignore the server's response and succeed with `initialModel`
- ⑦ We'll use this when the user clicks a photo to select it
- ⑧ The `SelectPhotoUrl` message sets `selectedPhotoUrl` in the model
- ⑨ Accept the new model we received from the server.
- ⑩ We'll ignore page load errors for now.

This gets our new page's `Model` started with a single field: `selectedPhotoUrl`, a `Maybe String` which is initially `Nothing`—indicating the user has not selected a photo yet—and which will be set to a particular photo's URL string if the user selects that photo.

We've also defined two `Msg` values.

1. The `SelectPhotoUrl` message sets the `selectedPhotoUrl` field in the model to be the URL contained in the message. We'll use this later, in the page's `view` function, when the user clicks a photo.
2. The `LoadPage` message contains data coming back from the server. Just like in Chapter 4, our `update` function will receive it once the HTTP request we specified in `init` either fails or successfully completes. If it completes, `update` will replace our `Model` with the contents of this message.

We can use what we have here to describe a page that loads photos from the server and then lets users click on them to select them. To complete this, we'll need to define a `view` function to render it all, as well as a `main` to tie the room together!

RENDERING THE NEW PAGE

Let's begin by adding a few imports to support our `view`:

```
import Html exposing (..)
import Html.Attributes exposing (class, src)
import Html.Events exposing (..)
```

Then let's add these `view` and `main` definitions to the end of `PhotoFolders.elm`:

```
view : Model -> Html Msg
view model =
    h1 [] [ text "The Grooviest Folders the world has ever seen" ]

main : Program Never Model Msg
main =
    Html.program { init = init, view = view, update = update, subscriptions = \_ -> Sub.none
    }
```

This won't display any actual photos yet, but it will at least get us to a point where we can bring up the page in a browser! We'll make this `view` function more featureful later.

VIEWING THE PAGE

Let's compile this by using a similar command to the one we've been running in the previous chapters, but this time with `PhotoFolders.elm` instead of `PhotoGroove.elm`:

```
elm make --output=elm.js PhotoFolders.elm
```

Now we can bring up the page in our browser to see how it looks so far. Before we can do this, we'll need to make one small tweak so that our new `PhotoFolders` code gets used instead of the `PhotoGroove` code like before. To do that, we'll need to replace the final `<script>` tag in `index.html` (the one that starts off `var app = ...`) with this much smaller one:

```
<script>
  var app = Elm.PhotoFolders.embed(document.getElementById("elm-area"));
</script>
```

NOTE Not only did the word `PhotoGroove` in this `var app =` declaration change to `PhotoFolders`, we also stopped passing `Pasta.version` and deleted all the code in the `<script>` related to ports.

Great! Save it and open the page in your browser, you should see something like Figure 7.2.



The Grooviest Folders the world has ever seen

Figure 7.2 Viewing the majestic new Photo Folders page

Glorious! Now we have a basic page up and running, and we're ready to start making it useful.

7.1.2 Storing Photos by URL in a Dictionary

As impressive as this page already is, it'll be even better when it does something! When the user selects a photo, we want to show a large version of it, along with some smaller thumbnails of its related photos. Figure 7.3 shows how we want it to look.

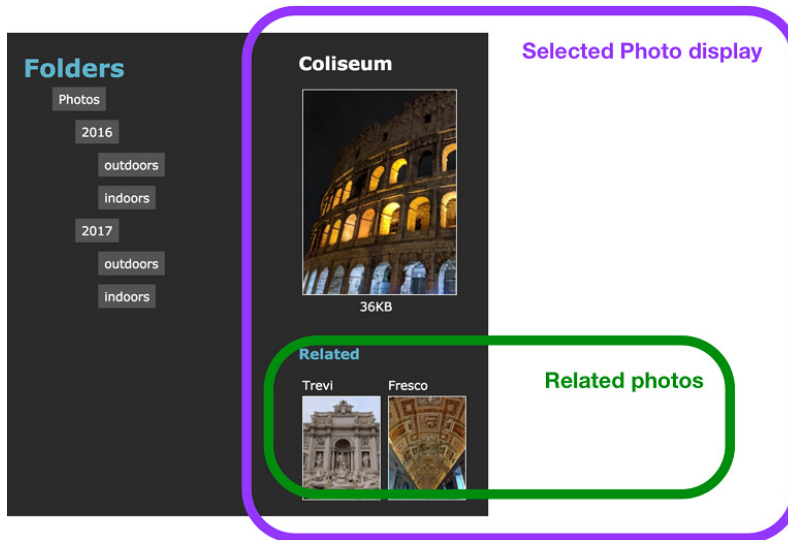


Figure 7.3 The Selected Photo and Related photos we'll be rendering

To implement this, we'll begin by creating two new functions, `viewSelectedPhoto` and `viewRelatedPhoto`, which we'll call from `view` to render the selected photo and its related thumbnails. Let's add the contents of Listing 7.2 to the end of `PhotoFolders.elm`.

Listing 7.2 PhotoFolders.elm

```
type alias Photo =
  { title : String
  , size : Int
  , relatedUrls : List String
  , url : String
  }

viewSelectedPhoto : Photo -> Html Msg
viewSelectedPhoto photo =
  div
    [ class "selected-photo" ]
    [ h2 [] [ text photo.title ]
    , img [src (urlPrefix ++ "photos/" ++ photo.url ++ "/full")] []
    , span [] [ text (toString photo.size ++ "KB") ]
    , h3 [] [ text "Related" ]
    , div [ class "related-photos" ]
      (List.map viewRelatedPhoto photo.relatedUrls)
    ]

viewRelatedPhoto : String -> Html Msg
viewRelatedPhoto url =
  img
```

```
[ class "related-photo"
, onClick (SelectPhotoUrl url)
, src (urlPrefix ++ "photos/" ++ url ++ "/thumb")
]
[]

urlPrefix : String
urlPrefix =
"http://elm-in-action.com/"
```

- ❶ We didn't have this field in `PhotoGroove.elm`
- ❷ Calling this function will require getting a `Photo` from our `Model` somehow
- ❸ Call `viewRelatedPhoto` on each URL in `photo.relatedUrls`
- ❹ This uses the `SelectPhotoUrl` message we defined earlier

We started by defining a `type alias Photo` much like the one we used in `PhotoGroove.elm`, except the one we defined before didn't have a `relatedUrls` field. (After all, the `RelatedPhotos` feature didn't exist back then!) We then used `Photo` to annotate `viewSelectedPhoto`.

NOTE: In Chapter 8 we'll address the code duplication between this file and `PhotoGroove.elm`, but in this chapter we'll focus only on the new page.

STORING AND RETRIEVING PHOTOS

Our new `viewSelectedPhoto` function takes a `Photo` as an argument, but our `Model` currently holds only a `Maybe String`. We'll need to begin storing some full-blown `Photo` records in our `Model` if we want to call that function!

Fortunately, our server will provide our `Model` with every `Photo` value we need when the page loads. This begs the question: once we obtain these `Photo` records from the server, how should we store them?

USING A LIST OF PHOTOS

One way would be to add a `photos : List Photo` field to our `Model`. Is a `List` the best choice we could make for this use case? Let's think through how `view` would use that `List Photo` to call `viewSelectedPhoto` : `Photo -> Html Msg`.

To call `viewSelectedPhoto` we'd take the selected URL from `model.selectedPhotoUrl` (assuming it's a `Just` rather than a `Nothing`) and look through every single record in our `List` in search of a `Photo` whose `url` field matches that selected URL. If we found a matching `Photo` record, we'd pass it to `viewSelectedPhoto`. Otherwise, we'd have nothing to render and would do nothing.

Although this approach works, it has two issues:

1. It's not terribly efficient for the task at hand. If the selected photo happens to be the final one in a list of a thousand photos, we have to examine every single one of those

photos, one at a time, to find it. It would be more efficient to use a data structure that didn't have to look through every single `Photo` in that situation!

2. In the event that multiple `Photo` records happen to have the given URL, it's not immediately clear how we will handle that.

USING A DICTIONARY OF PHOTOS

We can avoid both of these problems by storing the photos not in a list, but in a *dictionary*.

DEFINITION A *dictionary* is a collection where each *value* is associated with a unique *key*.

Back in Chapter 1, we talked about how **lists** in Elm must have elements of the same type; we can have `[1, 2]`, or `["a", "b"]`, but not `["a", 1]`. This is because lists can be iterated over—using functions like `List.map`—and reliable iteration depends on elements having consistent types. We also covered how **records** in Elm can store contents with mixed types, which comes at the cost of making it impossible to iterate over their fields or values.

Dictionaries have some of the characteristics of lists and some of the characteristics of records. Like lists, they can be iterated over. Like records, the values in a dictionary have no notion of a position, but we can efficiently look up any value by providing the associated key. Each key is guaranteed to be unique within the dictionary, so it is impossible to encounter multiple matches! This makes a dictionary a great choice for our use case.

Table 7.1 summarizes these trade-offs among the three data structures.

Table 7.1 Comparing Records, Lists, and Dictionaries

Structure	Iteration	Mixed Type Elements	Lookup by position	Lookup by key
Record	Unsupported	Supported	Unsupported	Efficient
List	Supported	Unsupported	Supported	Inefficient
Dictionary	Supported	Unsupported	Unsupported	Efficient

Let's get a better feel for dictionaries by playing around in `elm-repl`. Listing 7.3 shows how to create a dictionary and retrieve values from it using `Dict.fromList` and `Dict.get`.

Listing 7.3 Creating dictionaries and accessing their values

```
> import Dict
> dict = Dict.fromList [ ( "pi, give or take", 3.14 ), ( "answer", 42 ) ]
Dict.fromList [ ... ] : Dict String Float ❶

> Dict.get "a key we never added!" dict
Nothing : Maybe Float ❷

> Dict.get "pi, give or take" dict
Just 3.14 : Maybe Float ❸
```



```
> Dict.get "answer" dict
Just 42 : Maybe Float
```

④
④

- ① Dict String Float means “strings for keys, floats for values”
- ② Dict.get returns Nothing if the key is not found
- ③ Dict.get returns Just and the value if found
- ④ Dict.get on a Dict String Float always returns a Maybe Float

The `Dict.fromList` function takes a list of tuples and returns a `Dict`. Each tuple represents a key-value pair that will be stored in the dictionary.

If we call `Dict.get` on the resulting dictionary, passing a particular key, it will return the associated value—but only if it finds one! Remember how in Chapter 3 we saw that `Array.get` returns a `Maybe` value to represent the possibility that the requested element was not found? `Dict.get` does the same thing here: it returns `Nothing` if the given key was not present in the dictionary, and if the key was present, it returns the corresponding value wrapped in a `Just`.

TIP It's common for functions that look up values within Elm data structures to return a `Maybe` if the desired element might not be found. This approach is used in `Array.get`, `Dict.get`, `List.head`, and more!

DICT'S TYPE PARAMETERS

You may recall that `List` and `Array` are parameterized on one type—for example, `List String` or `Array Int`.

In contrast, `Dict` has two type parameters: one for the key, and one for the value—for example, `Dict Char Int` or `Dict String Photo`. In Listing 7.2 we had a `Dict String Float` because we passed a `List (String, Float)` to `Dict.fromList`, which meant this dictionary would have a `String` for each of its keys, and a `Float` for each of its values.

Constrained Type Variables: comparable, number, appendable

The type annotation for `Dict.get` is `(comparable -> Dict comparable value -> Maybe value)`. That `comparable` is no ordinary type variable! In Chapter 3 we noted that type variables can have *almost* any name you like. The exceptions responsible for that “almost” are the type variable names reserved for Elm’s *constrained* type variables, which behave differently from other type variables:

1. number, which can resolve to `Int` or `Float`
2. appendable, which can resolve to `String` or `List`
3. comparable, which can resolve to `Int`, `Float`, `Char`, `String`, `List` or a tuple of these

As an example of what these do, let’s look at the multiplication operator, `(*)`. If the type of `(*)` were `a -> a -> a`, you could use it to multiply anything—numbers, strings, you name it. However, its actual type is `number -> number -> number`. Because `number` is one of Elm’s constrained type variables, `number -> number -> number` can only resolve to one of the following:

```
Int -> Int -> Int
```

```
Float -> Float -> Float
```

The `number -> number -> number` function will resolve to one of these once it gets passed either an `Int` or a `Float` as one of its arguments. This means that multiplying two `Int` values will return an `Int`, multiplying two `Float` values will return a `Float`, and attempting to multiply an `Int` by a `Float` will result in a type mismatch! (The most common way to resolve this type mismatch is by calling `toFloat` on the `Int`.)

If you need to annotate a function which takes two independent `number` types, you can add more characters after the word `number` to create distinct type variable names that are still constrained by `number`. For example, `number`, `numberB`, `number2`, and `numberOfCats` all have different names but the same `number` constraint.

The `number` constraint typically appears in core functions having to do with mathematics, `appendable` is used by the `(++)` operator. `comparable` appears in the definitions of `Dict` and `Set`, as well as in `List.sort` and mathematical inequality operators like `(<)` and `(>=)`. Data structures like `Dict` and `Set` use the `comparable` constraint to store their contents efficiently, which involves sorting them—and only the `comparable` types have implicit sorting functions built in!

In the case of `Dict`, its keys must be `comparable`, but its values can be any type. We can have a `Dict String Photo` because `Photo` is the value type, which can be anything, and `String` is `comparable`, making it an acceptable key type. We could not have a `Dict Photo String`, though, because `Photo` is a type alias for a record, and neither records nor union types are `comparable`!

To create a dictionary with non-`comparable` keys, you can use a third-party package like `robertjlooby/elm-generic-dict`. Its API differs from `Dict` only in that `fromList`, `empty`, and `singleton` take one more argument.

ADDING A DICTIONARY TO MODEL

Let's introduce a dictionary of photos to our model. First we'll import the `Dict` module:

```
import Dict exposing (Dict)
```

Then we'll update `Model` to include a `Dict String Photo` field—with the `String` being the photo URL we'll use as the key, and the `Photo` being the corresponding value.

```
type alias Model =
  { selectedPhotoUrl : Maybe String
  , photos : Dict String Photo
  }
```

If we recompile, we'll get an error! `initialModel` now needs to include a `photos` field. We'll have it initialize that field to an empty dictionary using `Dict.empty` like so:

```
initialModel : Model
initialModel =
  { selectedPhotoUrl = Nothing
  , photos = Dict.empty
  }
```

Later in the chapter we'll override this empty dictionary with some `Photo` records loaded from the server, but for now having it start off empty will be enough to get things compiling again.

CALLING VIEWSELECTEDPHOTO FROM VIEW

Now we have a `viewSelectedPhoto` that does what we want, but only if `view` passes it a single `Photo` argument. How can `view` do that? The only argument `view` receives is `Model`, which has a `Dict String Photo` field and a `Maybe String` field, but no other sources of `Photo` values. How can we get a single `Photo` from this combination of values?

In situations like this, it's useful to take stock of the types we have to work with and the type we want to get to. It often turns out there are only a few functions or language features that do anything useful with those types!

Here's a summary of where we are:

- We **want** a `Photo` for the selected photo, so `view` can pass it to `viewSelectedPhoto`.
- We **have** a `Maybe String` indicating which photo is selected, if any.
- We **have** a `Dict String Photo` of all the photos.

How can we use what we **have** to get what we **want**?

USING DICT.GET WITH A MAYBE STRING

To get from a `Maybe String` of our selected photo's URL (assuming we actually have one) to a `Photo` record for that selected photo, we'll need to do three things:

1. Handle the possibility that `selectedPhotoUrl` is `Nothing`
2. If it wasn't `Nothing`, pass the selected photo URL to `Dict.get` on `model.photos`
3. Handle the possibility that `Dict.get` returned `Nothing`

We can do this with `Dict.get` and a nested *case-expression*, as shown in Listing 7.4:

Listing 7.4 Implementing `view` as a nested case-expression

```
selectedPhoto : Html Msg
selectedPhoto =
  case model.selectedPhotoUrl of
    Just url ->
      case Dict.get url model.photos of
        Just photo ->
          viewSelectedPhoto photo

        Nothing ->
          text ""

    Nothing ->
      text ""
```

- 1 Confirm `selectedPhotoUrl` is not `nothing`
- 2 Confirm that `url` is present in the dictionary
- 3 Empty text nodes render invisibly

REFACTORING TO USE `MAYBE.ANDTHEN`

We can express this same logic more concisely using the `Maybe.andThen` function:

```
Maybe.andThen : (original -> Maybe final) -> Maybe original -> Maybe final
```

Whenever we have two nested *case-expressions* like this, where both of them handle `Nothing` the same way, `Maybe.andThen` does exactly the same thing as the code we had before!

We'll use `Maybe.andThen` when we define `selectedPhoto` in our actual `view` function. Let's replace our existing `view` function with this definition:

Listing 7.5 Implementing view using Maybe.andThen

```
view : Model -> Html Msg
view model =
  let
    photoByUrl : String -> Maybe Photo
    photoByUrl url =
      Dict.get url model.photos

    selectedPhoto : Html Msg
    selectedPhoto =
      case Maybe.andThen photoByUrl model.selectedPhotoUrl of
        Just photo ->
          viewSelectedPhoto photo

        Nothing ->
          text ""

  in
  div [ class "content" ]
    [ div [ class "selected-photo" ] [ selectedPhoto ] ]
```

- ❶ Returns the photo with the given URL in `model.photos`
- ❷ `photo` is the selected photo, if found in `model.photos`
- ❸ finish view by wrapping `selectedPhoto` in the structure the page needs

Comparing Maybe.andThen and Maybe.map

`Maybe.andThen` works a bit like the `Maybe.map` function we saw in Chapter 6. Here's how their types compare:

```
Maybe.andThen : (original -> Maybe final) -> Maybe original -> Maybe final
Maybe.map     : (original ->      final) -> Maybe original -> Maybe final
```

Both functions accept a callback and a `Maybe`, both return `Nothing` when given `Nothing`, and when given a `Just`, both will pass the value inside that `Just` to their callbacks.

Where they differ is that if `Maybe.map` receives a `Just`, it always returns a `Just`. In contrast, `Maybe.andThen` returns whatever its callback returns—which means it can potentially receive a `Just` and turn it into a `Nothing`!

This makes `andThen` strictly more powerful than `map`. Anything we can implement with `map`, we could implement with `andThen` instead! However, `Maybe.map` tends to see more use in practice because it's more concise, and more often than not, the extra power of `Maybe.andThen` is not needed.

RENDERING THE SELECTED PHOTO

Now everything compiles, but we can't see if it's working properly because we initialize our photo dictionary to `Dict.empty` and never change it. If we pretend we're already able to decode some useful `Photo` data from the server, we'll be able try out our new page!

Let's change `modelDecoder` to this:

```
modelDecoder : Decoder Model
modelDecoder =
  Decode.succeed
    { selectedPhotoUrl = Just "trevi"
    , photos = Dict.fromList
      [ ( "trevi", { title = "Trevi", relatedUrls = [ "coli", "fresco" ], size = 34,
        url ="trevi" } )
      , ( "fresco", { title = "Fresco", relatedUrls = [ "trevi" ], size = 46, url
        ="fresco" } )
      , ( "coli", { title = "Coliseum", relatedUrls = [ "trevi", "fresco" ], size =
        36, url ="coli" } )
      ]
    }
```

If we re-run `elm make --output=elm.js PhotoFolders.elm` and open the result in the browser, we can now see a lovely large photo with some related photos below it.

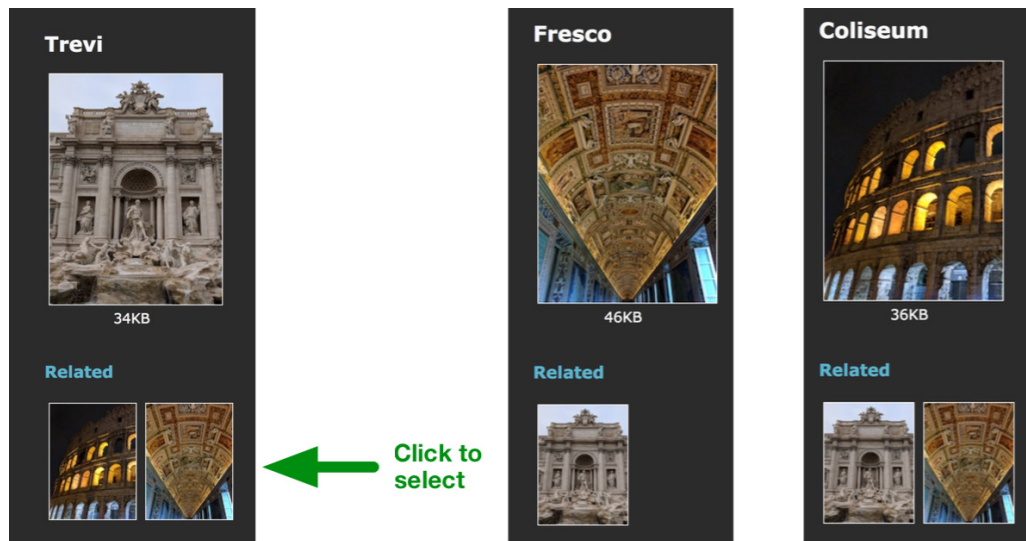


Figure 7.4 Viewing the selected photo with related photos below it

Fantastic! Not only does this page display the related photos below the selected one, but we can click them to change which one is selected. Next we'll introduce the *Pholders*—er, *Folders*!

7.2 Modeling Trees using Recursive Union Types

Since another of our coworkers is taking care of the page where users edit their folders, we are free to focus on the page that lets users browse them. Later in the chapter we'll load the folders and their contents from the server, but we'll start by hardcoding them just like we did with the Related Photos. This way we can get the interaction working to our satisfaction before we have to deal with whatever data format the server happens to give us!

Each folder can potentially contain one or more photos, as well as one or more subfolders—each of which can also contain more photos and more subfolders, and so on. This relationship forms a *tree*, which is a data structure we've never used in the previous chapters!

7.2.1 Defining Trees using union types

Let's say we tried to represent our folder tree structure with a type alias, like so:

```
type alias Folder = { name : String, subfolders : List Folder }
```

This won't work! If we try to compile it, we'll get an error. What's the problem with it?

The purpose of `type alias` is give a name to a particular type annotation. When the compiler encounters this name, it substitutes in the type annotation we associated with that name and proceeds as normal.

Now let's suppose the compiler encounters a `Folder` annotation and begins to expand it into the complete type. What happens? The expansion would go something like this:

```
{ name: String, photoUrls : List String, subfolders : List
  { name : String, photoUrls : List String, subfolders : List
    { name : String, photoUrls : List String, subfolders : List ...
```

Uh oh. Trying to expand that type annotation would never end! This is why defining a `type alias` in terms of itself doesn't work, and why we'll need to use something else to define this `Folder` type.

RECURSIVE UNION TYPES

Fortunately, where type aliases give a name to an existing type, union types actually define a brand new type—and they can refer to themselves in their own definitions! Union types which do this are known as a *recursive* union types.

One of the many data structures we can define using a recursive union type is a linked list—such as the `List` type we all know and love. Under the hood, an Elm `List` is structured like this union type:

```
type MyList elem
  = Empty
  | Prepend elem (MyList elem)
```

NOTE When we write `[1, 2, 3]`, it's essentially syntax sugar for `Prepend 1 (Prepend 2 (Prepend 3 Empty))`. No other language support is necessary; every `List` function we've been using in this book—`map`, `head`, and so on—can be built using nothing more than *case-expressions* on this union type!

DEFINING FOLDER AS A RECURSIVE UNION TYPE

We can use a similar approach to define `Folder` as a recursive union type like so:

```
type Folder =
  Folder
    { name : String
    , photoUrls : List String
    , subfolders : List Folder
    }
```

Let's add this `type Folder` declaration right above our `type alias Model` declaration in `PhotoFolders.elm`.

Notice that whereas other union types we've seen have had multiple constructors—for example, `type Msg = SelectByUrl String | SetFilter String Int`—this union type has only one constructor! It holds plenty of information, though, because that one constructor contains a record: `type Folder = Folder { name : String, ... }`.

This is a common technique in Elm: when a `type alias` is the wrong fit, upgrading to a union type with a single constructor. It's typical when doing this to give the single constructor the same name as the type itself—in this case, `type Folder = Folder { ... }`—but we could have just as easily called it something like `type Folder = SingleFolder { ... }` instead.

NOTE TO MEAP READERS: the following Tip applies to the next release of Elm (version 0.19); `-optimize` doesn't exist in the 0.18 compiler.

TIP This upgrade has no runtime cost when you run `elm make` with the `--optimize` flag. When Elm's compiler sees a union type with a single constructor, it “unboxes” it, such that `type Foo = Foo String` compiles down to a plain `String` at runtime.

ADDING HARDCODED DATA TO MODEL AND INIT

Now that we've defined `Folder`, let's make these changes to `Model` and `init`:

1. Add a `root : Folder` field to the end of our `type alias Model` declaration. This will represent our root folder, which will contain all the subfolders inside it.
2. Add `root = Folder { name = "Loading...", photoUrls = [], subfolders = [] }` to the end of the `initialModel` record. This defines a placeholder root folder that says "Loading..." while we wait for the server to respond with the actual folders.

NOTE It would be nicer to model this using something like a `Maybe Folder` to indicate that the data has not loaded yet. In Chapter 8 we'll refactor to an approach more like that!

Let's also give ourselves some example folders to work with, by adding a hardcoded root folder to our `modelDecoder` like so:

```
modelDecoder : Decoder Model
modelDecoder =
  Decode.succeed
    { selectedPhotoUrl = ...
    , photos = ...
    , root =
      Folder { name = "Photos", photoUrls = [], subfolders = [
        Folder { name = "2016", photoUrls = [ "trevi", "coli" ], subfolders = [
          Folder { name = "outdoors", photoUrls = [], subfolders = [] },
          Folder { name = "indoors", photoUrls = [ "fresco" ], subfolders = [] }
        ]},
        Folder { name = "2017", photoUrls = [], subfolders = [
          Folder { name = "outdoors", photoUrls = [], subfolders = [] },
          Folder { name = "indoors", photoUrls = [], subfolders = [] }
        ]}
      ]}
    }
```

Now we can access `model.root` from our `view` function, and use it to render some folders!

RENDERING FOLDERS

To render the folders, we'll start by writing a `viewFolder` function that renders a single folder. Let's add the contents of Listing 7.4 right after `viewRelatedPhoto`.

Listing 7.4 viewFolder

```
viewFolder : Folder -> Html Msg
viewFolder (Folder folder) = ❶
  let
    subfolders =
      List.map viewFolder folder.subfolders ❷
  in
    div [ class "folder" ]
      [ label [] [ text folder.name ]
      , div [ class "subfolders" ] subfolders
      ]
❶ Inline pattern match
❷ viewFolder calls itself
```

This `viewFolder` implementation demonstrates a couple of new tricks!

DESTRUCTURING SINGLE-CONSTRUCTOR UNION TYPES

First up is that `viewFolder (Folder folder)` syntax. This is a syntax shorthand that makes our code more concise. Our `Folder` type is a union type which holds a single constructor, and inside that constructor is a record we want to access (with `{ name : String }` and so on).

One way we could access that record is using a *case-expression* to destructure our union type's one and only constructor (the constructor named `Folder`, just like the type itself) the way we did in Chapter 3:

```
viewFolder : Folder -> Html Msg
viewFolder wrappedFolder =
  case wrappedFolder of
    Folder folder ->
      ..."folder" now refers to the record we want...
```

Inside that one branch of the *case-expression*, `folder` would refer to the record we want, and we could then use it however we pleased. There's nothing wrong with this code, except that it could be shorter without losing any clarity!

Since the `Folder` union type has exactly one constructor, Elm lets us avoid writing a full *case-expression* by using the shorthand `viewFolder (Folder folder) =` to destructure it inline instead. That shorter version is equivalent to the longer *case-expression* version above.

RECURSIVE FUNCTIONS

Next we have this expression: `List.map viewFolder folder.subfolders`. Notice that `viewFolder` is calling itself, making it a *recursive* function.

DEFINITION: A function that calls itself is known as a *recursive* function.

INCORPORATING VIEWFOLDER INTO VIEW

Let's use our hardcoded `Model` data to try out this `viewFolder` function. Add a `div [class "folders"]` right before `div [class "selected-photo"]`, with the following inside it:

```
div [ class "content" ]
  [ div [ class "folders" ]
    [ h1 [] [ text "Folders" ]
      , viewFolder model.root
    ]
  , div [ class "selected-photo" ] [ selectedPhoto ]
  ]
```

Let's recompile and check out the results of our efforts in the browser:

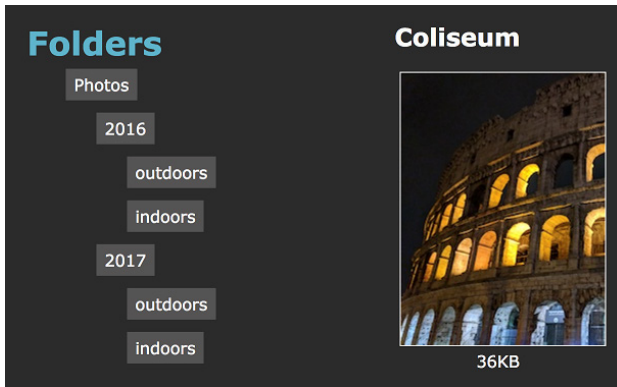


Figure 7.5 Viewing folders

Looking good! We can see our hardcoded folders rendered to the screen, but clicking on them still does nothing at this point. Next we'll introduce the ability to expand and to collapse them!

7.2.2 Recursive messages

Whenever the user clicks a folder, we want it to toggle between being expanded and collapsed. This means we'll need to introduce a new piece of state to our `Folder`: a `Bool` indicating whether the folder is expanded. Let's start by making these changes:

1. Add an `expanded : Bool` field to the end of the record in our type `Folder` definition.
2. Add `expanded = True` to each of the `Folder` records in our hardcoded `modelDecoder`. (A quick way to do this is to find/replace all instances of `"photoUrls ="` with `"expanded = True, photoUrls ="` in our `PhotoFolders.elm` file.)

DESCRIBING DESCENT

To implement this feature we'll need to add an `onClick` handler to each folder, to toggle its expanded state. Here's how we'll do this:

1. Pass `onClick` a new `Msg` that tells `update` which folder the user clicked.
2. When `update` receives one of these message, descend from the `root` folder into the folder tree and toggle the appropriate folder's `expanded` field in the `Model`.

Listing 7.5 shows the code we'll add to the end of `PhotoFolders.elm` to accomplish this:

Listing 7.5 toggleExpanded and FolderPath

```
type FolderPath
  = End
  | Subfolder Int FolderPath
```

1

2

```

toggleExpanded : FolderPath -> Folder -> Folder
toggleExpanded path (Folder folder) =
  case path of
    End ->
      Folder { folder | expanded = not folder.expanded }

    Subfolder targetIndex remainingPath ->
      let
        subfolders : List Folder
        subfolders =
          List.indexedMap transform folder.subfolders

        transform : Int -> Folder -> Folder
        transform currentIndex currentSubfolder =
          if currentIndex == targetIndex then
            toggleExpanded remainingPath currentSubfolder
          else
            currentSubfolder
      in
        Folder { folder | subfolders = subfolders }

```

- ❶ A path from a folder to a particular subfolder
- ❷ Like `Folder`, `FolderPath` is also a recursive union type
- ❸ Destructuring the `Folder` union type inline
- ❹ ``not`` swaps `True` for `False`, and vice versa
- ❺ `targetIndex` is an `Int`. `remainingPath` is a `FolderPath`
- ❻ `toggleExpanded` is recursive; it calls itself

`toggleExpanded` takes a `FolderPath` and a `Folder`, and does one of the following:

1. If `FolderPath` is `End`, there are no subfolders to traverse into, so toggle the `expanded` value on the given folder.
2. If `FolderPath` is `Subfolder targetIndex`, look through the given root's subfolders until we find the one at position `targetIndex`. Then call `toggleExpanded` again, this time passing that subfolder as the new root folder, and passing the remaining `FolderPath` after discarding the `Subfolder` value we just handled.

Technically, the function only ever toggles the “root folder”, but its notion of which folder is the root changes as it processes `Subfolder` values.

Once it has resolved all the subfolders, the final call to `toggleExpanded` will receive the originally desired subfolder as its “root” argument, along with `Root` as its `FolderPath`, meaning it will toggle the `expanded` value like we want!

Figure 7.6 shows how `FolderPath` can model a path to whichever folder the user clicked, and how `toggleExpanded` recurses along that path.

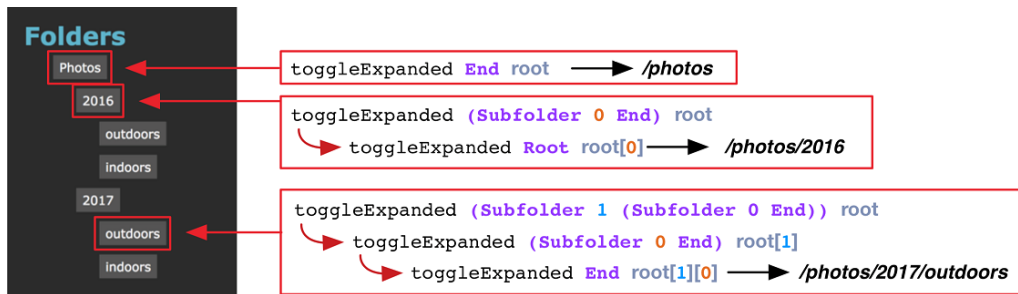


Figure 7.6 Using `toggleExpanded` to represent user interaction with folders

LIST.INDEXEDMAP

To update only the subfolder at the desired index, `toggleExpanded` makes use of `List.indexedMap`, a function we haven't encountered yet! It works similarly to `List.map`, but with one difference. Here are their types side by side:

```
List.map      : (oldVal -> newVal) -> List oldVal -> List newVal
List.indexedMap : (Int -> oldVal -> newVal) -> List oldVal -> List newVal
```

The only thing `List.indexedMap` does differently from `List.map` is that it passes an additional value to the transformation function: an `Int` representing the element's index within the list.

If you call `List.indexedMap` on the list `["foo", "bar", "baz"]`, the transformation function will receive 0 and "foo", then 1 and "bar", and finally 2 and "baz".

TRANSFORMING SUBFOLDERS BASED ON INDEX

Here's how `toggleExpanded` uses `List.indexedMap` to alter subfolders.

```
subfolders : List Folder
subfolders =
  List.indexedMap transform root.subfolders
```

When `transform` gets called by `indexedMap`, it receives not only the particular subfolder it's about to transform, but also that subfolder's index within the `subfolders` list.

It makes use of that index like so:

```
transform : Int -> Folder -> Folder
transform currentIndex currentSubfolder =
  if currentIndex == targetIndex then
    toggleExpanded remainingPath currentSubfolder
  else
    currentSubfolder
```

Since the `else` branch returns the original subfolder unchanged, this `transform` function only actually transforms subfolder if `currentIndex == targetIndex`. It transforms the subfolder at the requested `targetIndex`, and leaves the others alone.

INCORPORATING FOLDERPATH INTO MSG

Next we'll connect `toggleExpand` to `update` by expanding our `Msg` type, introducing a `ToggleExpanded` constructor to go with our existing `SelectPhotoUrl` and `LoadPage` constructors. This new constructor will fire whenever the user clicks a folder, and it will use a `FolderPath` to describe where in the folder hierarchy the user clicked.

Listing 7.3 shows the changes we'll make to `Msg` and `update`.

Listing 7.3 Adding ToggleExpanded to Msg and update

```
type Msg
  = SelectPhotoUrl String
  | LoadPage (Result Http.Error Model)
  | ToggleExpanded FolderPath ❶

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    ToggleExpanded path ->
      ( { model | root = toggleExpanded path model.root }, Cmd.none ) ❷
    SelectPhotoUrl url -> ...
```

- ❶ We'll make this fire when a user clicks a folder
- ❷ Add a branch to cover our new `Msg` constructor

Next we'll make things interactive by connecting this `update` logic to `view`!

7.2.3 Event handlers with recursive messages

Now that `update` supports a `Msg` called `ToggleExpanded` which toggles the expanded state of a particular folder, we can modify our `viewFolder` function to make the folders toggle between being expanded and collapsed on click. While we're at it, we can also start rendering the folders differently depending on whether they are `expanded`!

Listing 7.4 shows how we'll make both of these changes.

Listing 7.4 viewFolder with toggling expanded

```
viewFolder : FolderPath -> Folder -> Html Msg
viewFolder path (Folder folder) =
  let
    viewSubfolder : Int -> Folder -> Html Msg
    viewSubfolder index subfolder =
      viewFolder (appendIndex index path) subfolder ❶ ❷

    folderLabel =
      label [ onClick (ToggleExpanded path) ] [ text folder.name ] ❸ ❹

  in
    if folder.expanded then
      let
        contents = ❺
```

```

        List.indexedMap viewSubfolder folder.subfolders ③
    in
    div [ class "folder expanded" ]
        [ folderLabel ④
          , div [ class "contents" ] contents
        ]
    else
    div [ class "folder collapsed" ] [ folderLabel ] ⑤

appendIndex : Int -> FolderPath -> FolderPath
appendIndex index path =
    case path of
        End ->
            Subfolder index End ⑥

        Subfolder subfolderIndex remainingPath ->
            Subfolder subfolderIndex (appendIndex index remainingPath) ⑦

```

- ① Each subfolder's `FolderPath` is nested one deeper
- ② Click the folder's name to toggle expanded
- ③ Render each subfolder using its index
- ④ Render the folder's label before its contents
- ⑤ Don't render contents of collapsed folders
- ⑥ Replace the original `End` with `(Subfolder index End)`
- ⑦ recurse until we reach the `End`

APPENDINDEX

The `appendIndex` function is another recursive function! (As it turns out, they come up a lot when working with tree structures.) This one adds a new subfolder index onto the end of a `FolderPath`, so that `viewFolder` can build up a `FolderPath` as it works its way from the root folder through the subfolders.

NOTE If we wrote `(Subfolder index path)` instead of `(appendIndex index path)` inside `viewFolder`, the folder paths would get reversed! This wouldn't make a noticeable difference for the root folder and its first subdirectories, but it would result in some funky behavior when toggling other subfolders.

MODIFYING VIEW

We'll also need to modify our top-level `view` function to change how it renders the root folder:

```
viewFolder End model.root
```

Thanks to our new `view` function, a `ToggleExpanded` message will be sent to `update` whenever the user clicks a folder. Figure 7.7 shows the exact `Msg` that will be sent to `update` when each folder is clicked.

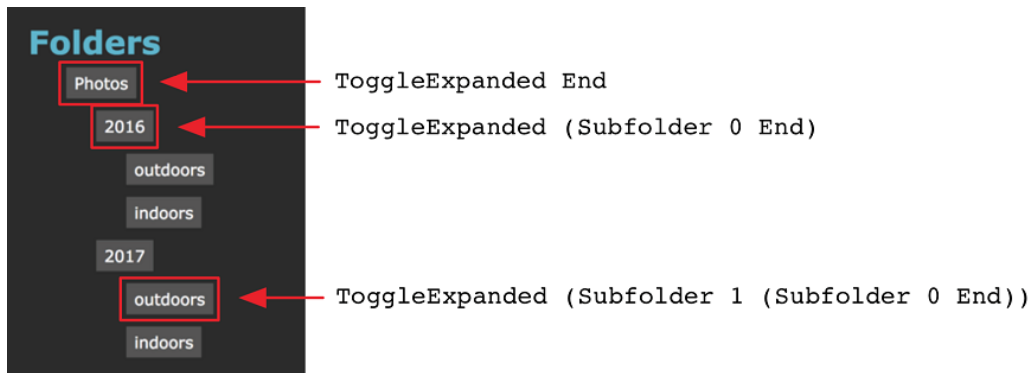


Figure 7.7 User clicks a subfolder and `ToggleExpanded` gets sent to update

Great! We only have one step remaining: make it so when the user clicks a photo, it becomes the selected one.

SELECTING A PHOTO ON CLICK

First we'll introduce a `viewPhoto` function above `viewSelectedPhoto`, which will render an individual photo within a folder. We'll have this function reuse our existing `SelectPhotoUrl` message from section 7.1.1 like so:

```
viewPhoto : String -> Html Msg
viewPhoto url =
  div [ class "photo" , onClick (SelectPhotoUrl url) ]
    [ text url ]
```

Next we'll use `viewPhoto` to render the photos right below where we're rendering the subfolders. Let's change the definition of `contents` inside `viewFolder` to look like this:

```
contents =
  List.append
    (List.indexedMap viewSubfolder folder.subfolders)
    (List.map viewPhoto folder.photoUrls)
```

NOTE: If you want an extra challenge, at the end of this chapter try introducing a feature where clicking the Related Photo causes the tree to expand to that photo!

Now you can click either a photo within its folder or in the Related Photos list, and have it become the selected photo. We've made things nicely interactive!

7.3 Decoding Graphs and Trees

We've now built an interface which renders a tree of folders, each of which can contain photos and subfolders, as well as a Selected Photo display which includes Related Photos. We defined

hardcoded data for them so that we could try them out, but now it's time to load the data for these displays on the fly—by decoding JSON from our server!

7.3.1 Decoding Dictionaries

The `Model` we defined in Section 7.1 stores its photos in a field called `photos` : `Dict String Photo`. We'll start by defining a `Decoder` (`Dict String Photo`) which can decode one of those dictionaries from JSON.

SAMPLE JSON

Here's an example of the JSON format our server will send us to represent the photos:

```
{
  "2turtles":    {"title": "Two Turtles ",      "related_photos": ["beach"],
                  "size": 27},
  "beach":       {"title": "At Chang's Beach!", "related_photos": ["wake",
                  "2turtles"], "size": 36},
  "wake ":      {"title": "Boat wake",          "related_photos": ["beach "],
                  "size": 21}
}
```

In this structure, each photo's URL is a field in a JSON object, and the value associated with that URL holds the rest of the information about that photo. Translating this JSON into a `Photo` record will require decoding techniques that go beyond the basics we learned in Chapter 4!

COMPARING PHOTO REPRESENTATIONS

First, let's compare how a `Photo` will be represented in our Elm code with the corresponding JSON representation coming from the server.

Table 7.4 Comparing type alias `Photo` and the JSON object we get from the server

Elm Record	JSON Object
<pre>type alias Photo = { title : String , size : Int , relatedUrls : List String , url : String }</pre>	<pre>{ "title": "Chang's Beach", "size": 24, "related_photos": ["maui.jpg"] }</pre>

INTERMEDIATE REPRESENTATIONS

The JSON object has three fields: `title`, `size`, and `related_photos`. That's not enough data for us to build a complete `Photo` record, but we can at least write a decoder that handles 3 of the 4 fields. We can figure out how to incorporate the fourth one afterwards!

Let's begin by adding the code in Listing 7.5 to the end of `PhotoFolders.elm`.

NOTE TO MEAP READERS: In Chapter 4 we used ``decode`` instead of ``succeed`` here. ``decode`` was an alias for ``succeed``, but it has recently been deprecated in favor of using ``succeed`` directly. The next revision of Chapter 4 will use ``succeed``, so this chapter will do that as well!

Listing 7.5 Creating a `jsonPhotoDecoder`

```
type alias JsonPhoto =
  { title : String
  , size : Int
  , relatedUrls : List String
  }

jsonPhotoDecoder : Decoder JsonPhoto
jsonPhotoDecoder =
  Decode.succeed JsonPhoto
    |> required "title" string
    |> required "size" int
    |> required "related_photos" (list string)
```

- ❶ A type alias to represent the Photo info we get from JSON
- ❷ Decode these fields from the JSON object into a `JsonPhoto` record
- ❸ Like we saw in Chapter 4, section 4.3.2, this decodes a `List String` into `JsonPhoto`'s third field: `relatedUrls`

The `JsonPhoto` type we've introduced here is an *intermediate representation*—a value we'll use only to help us translate from one value to another. In particular, a `JsonPhoto` value will help us get from JSON to a `Photo` record.

DECODING URL STRINGS FROM ENCLOSING OBJECTS

We can now decode a `JsonPhoto` record, which holds all the information necessary to make a `Photo` record except a URL string. If we can find some way to decode a URL string, we could combine that with our `JsonPhoto` to end up with the `Decoder Photo` we want!

With that in mind, let's take another look at our sample JSON:

```
{ "2turtles.jpg": { ... }, "beachday.jpg": { ... }, "day1maui.jpg": { ... } }
```

Hey, there are the URLs we need, right in the object's keys!

DECODE.KEYVALUEPAIRS

We can decode those keys using the `Decode.keyValuePairs` function. It has this type:

```
keyValuePairs : Decoder val -> Decoder (List ( String, val ))
```

This gives us a decoder that translates JSON objects into key-value tuples. The key's type is always `String`, because JSON object keys are strings by definition. The value's type depends on the `Decoder` we pass to the `keyValuePairs` function.

As an example, suppose we called `keyValuePairs jsonPhotoDecoder` and ran the resulting decoder on our JSON sample from earlier:

```
{ "turtles.jpg": { ... }, "beach.jpg": { ... }, "maui.jpg": { ... } }
```

The output would be a list of (`String`, `JsonPhoto`) tuples like so:

```
output : List ( String, JsonPhoto )
output =
  [ ( "2turtles.jpg", { title = "Turtles & sandals", ... } )
  , ( "beachday.jpg", { title = "At Chang's Beach!", ... } )
  , ( "day1maui.jpg", { title = "First day on Maui", ... } )
  ]
```

TRANSLATING THE LIST OF TUPLES INTO A DICTIONARY

Now we can decode a list of (`String`, `JsonPhoto`) tuples, where the `String` is a photo's URL and `JsonPhoto` is everything else but the URL. We're getting closer!

Let's write a function that translates that list into the dictionary we want, by converting each (`String`, `JsonPhoto`) tuple into a (`String`, `Photo`) tuple and then passing a list of those to `Dict.fromList`. Here's the code we'll add to the end of `PhotoFolders.elm` to do this:

```
finishPhoto : ( String, JsonPhoto ) -> ( String, Photo )
finishPhoto ( url, json ) =
  ( url
  , { url = url
    , size = json.size
    , title = json.title
    , relatedUrls = json.relatedUrls
    }
  )

fromPairs : List ( String, JsonPhoto ) -> Dict String Photo
fromPairs pairs =
  pairs
    |> List.map finishPhoto
    |> Dict.fromList
```

Just like we did back in Section 7.1.2 with `viewSelectedPhoto`, let's once again take stock of the values we **have**, and see if we can combine them to get the value we **want**.

- We **want** a `Decoder (Dict String Photo)` where each key is the `Photo`'s url.
- We **have** a `Decoder (List (String, JsonPhoto))` where each `String` is the URL that goes with that `JsonPhoto`.

How can we use what we **have** to get what we **want**?

COMBINING THE INGREDIENTS WITH `DECODE.MAP`

Just like before, there are often only a few functions or language features that do anything useful with these types! In this case, we can reach our goal using the `Decode.map` function, whose type may look familiar to fans of related artists `List.map`, `Maybe.map`, and `Result.map`:

```
Decode.map : (original -> goal) -> Decoder original -> Decoder goal
```

Here, our original type is `List (String, JsonPhoto)` and our goal type is `Dict String Photo`. Do we have a function that takes the one and returns the other?

We sure do! The `fromPairs` function we just wrote does exactly that. (What are the odds?) Let's add this to the end of `PhotoFolders.elm`, so that `modelDecoder` can use it later.

```
photosDecoder : Decoder (Dict String Photo)
photosDecoder =
  Decode.keyValuePairs jsonPhotoDecoder
    |> Decode.map fromPairs
```

TIP This “How can we use what we have to get what we want?” technique works out quite often! It's a useful approach beyond just the two examples we've seen in this chapter, with views and with JSON decoding.

Our quest for a `Decoder (Dict String Photo)` is at an end. Huzzah!

7.3.2 Decoding recursive JSON

Now that we've added a JSON decoder for our `Photo` record, all we need is to do the same for `Folder`, and we should be able to load a complete `Model` from our server! Let's see what JSON the server gives us for folders.

RECURSIVE JSON

Figure 7.8 shows a sample of the JSON we'll receive from the server to describe our folders.

```
{
  "name": "All Photos",
  "photos": { "2turtles": {title: "Turtles & sandals", ... } },
  "subfolders": [
    { "name": "2016", "photos": {...}, "subfolders": [...] },
    { "name": "2017", "photos": {...}, "subfolders": [...] }
  ]
}
```

our `Decoder (Dict String Photo)` decodes this

each subfolder has the same structure as this outer **Folder**

Figure 7.8 Sample of the JSON we will get back from the server to describe a Folder.

Let's focus on the outermost JSON object, which has this structure:

```
{"name": <folder name>, "photos": <photos>, "subfolders": <subfolders> }
```

Each of the `<subfolders>` also has this structure, and so do the subfolders' subfolders, and so on all the way down. It's a *recursive* JSON structure! Thankfully, our `Folder` type is already recursive, so it'll be able to store the data this JSON describes.

There's a catch: this is the one and only JSON structure the server will ever give us. That means we'll need to assemble our entire `Model` using only JSON in this format!

DECODING THE COMPLETE MODEL

Can this possibly be enough data to decode a complete `Model`?

For that, we need values for the `Model`'s `root : Folder` field as well as its `photos : Dict String Photo` field. Also, the `Dict String Photo` needs to be a single source of truth for all photos across all subfolders; otherwise we won't be able to click any photo in any folder to select it.

But where is our single source of truth in this JSON? Sure, each subfolder has its own `photos` fields containing some `Photo` values, but that's not a single source of truth! Technically this JSON contains all the information we need, but it's not organized in our ideal format; the photo data is intermixed with the folder data, whereas we want them to be neatly separated.

GATHERING NESTED PHOTOS

We ask the team who maintains this server endpoint if they'd be willing to make changes to their code to provide the JSON in our preferred format, but they're swamped lately—something about a “primary key overflow,” whatever that means—and can't accommodate us.

Instead of contorting the lovely data model we've built up over the course of the chapter to fit this JSON we happened to get from the server, we'll write a decoder that translates it into the format we want. After all, the whole point of decoders is to decouple our application's data model from whatever curveball external data formats the world decides to throw at us.

In a different situation, we could be accessing a third-party service where we have no say whatsoever in how the data gets formatted!

DECODING A FOLDER

Let's write a decoder for our root `Folder` first. Here's the sample JSON we'll be working with:

```
{"name": "pics from 2016", "photos": { ... }, "subfolders": [ ... ]}
```

We want to translate this into a `Folder` value where:

1. The folder's `name` is `"pics from 2016"`
2. The folder's `photoUrls` value corresponds to the keys in that `"photos"` object
3. Each of the folder's subfolders is represented the same way this one was

To do this, we'll add the contents of Listing 7.5 to the end of `PhotoFolders.elm`.

Listing 7.5 Decoding both folders and photos in a single decoder

```
folderDecoder : Decoder Folder
folderDecoder =
  Decode.succeed folderFromJson
    |> required "name" string
    |> required "photos" photosDecoder
    |> required "subfolders" (list folderDecoder)
```

①

②

③

```

folderFromJson : String -> Dict String Photo -> List Folder -> Folder ④
folderFromJson name photos subfolders =
  Folder
    { name = name
    , expanded = True ⑤
    , subfolders = subfolders
    , photoUrls = Dict.keys photos ⑥
    }

```

- ① On success, pass decoded name, photos, and subfolders to `folderFromJson`
- ② We defined `photosDecoder` in Section 7.3.1
- ③ Yikes! This decoder is defined in terms of itself!
- ④ These argument types match the `require` calls above
- ⑤ We'll expand each folder by default
- ⑥ `photos` is a `Dict String Photo`; its keys are photo URLs

Let's compare how we built our `Decoder Photo` to how we've built a `Decoder Folder` here.

Here are the steps we used to create our `Photo` decoder:

1. We used type alias `JsonPhoto` as an intermediate representation, which gave us a `JsonPhoto : String -> Int -> List String -> JsonPhoto` function.
2. We called `Decode.succeed JsonPhoto`, meaning this `JsonPhoto` function would be called if all the decoding steps succeeded.
3. We used `required` several times, to build up a `Decoder JsonPhoto` by specifying the fields and types we wanted to decode and then pass to the `JsonPhoto` function, which then returns a decoded `JsonPhoto` value.
4. We used `Decode.map` to translate a `Decoder JsonPhoto` into the `Decoder Photo` we ultimately wanted.

Here we're using a different approach to get a similar outcome:

1. Instead of using an intermediate representation like `JsonPhoto`, we handcrafted a `folderFromJson : String -> Dict String Photo -> List Folder -> Folder` function.
2. We called `Decode.succeed folderFromJson`, meaning this `folderFromJson` function will be called if all the decoding steps succeed.
3. We used `required` several times, to build up a `Decoder Folder` by specifying the fields and types we wanted to decode and then pass to the `folderFromJson` function, which returns a decoded `Folder` value.

NOTE With this approach, we did not use an intermediate representation like `JsonPhoto`, so we did not need to use `Decode.map` to translate between that representation and the final decoder type we wanted.

This decoder ends up doing a bit more “post-processing” than other decoders we've written—namely calling `Dict.keys` on `photos` to get a `List String` of its keys, and adding `expanded = True`—but not much!

IDENTIFYING A CYCLIC DEFINITION

Our `folderDecoder` implementation from Listing 7.5 almost works...but not quite! If we try to compile it, we'll get a `BAD RECURSION` error.

The trouble is that we've defined a value in terms of itself. Imagine if we'd written this:

```
myString : String
myString =
  List.reverse myString
```

This has a similar problem to the one we saw in Section 7.2.1 when we tried to make a `type alias Folder` that referenced `Folder` in one of its fields. Since a type alias is a shorthand for a type we could have written by hand, when the compiler went to expand that alias out to the full type, it discovered that the resulting type would never end!

Just like how a `type alias` declaration names a *type*, this `myString =` declaration names an *expression*. Any time the compiler encounters `myString`, it will substitute the expression after the equals sign. This is where things go wrong.

Let's look at how the compiler would expand `myString`:

```
List.reverse (List.reverse (List.reverse (List.reverse ...
```

The expansion never ends, because the compiler substitutes in `List.reverse str` as soon as it sees `myString`, then sees a `myString` in that `List.reverse myString` expression, and therefore substitutes `List.reverse myString` into *that* expression...and so on forever.

THE CYCLIC DEFINITION IN FOLDERDECODER

Our current `folderDecoder` definition has this same problem. Here it is again:

```
folderDecoder : Decoder Folder
folderDecoder =
  Decode.succeed folderFromJson
    |> required "name" string
    |> required "photos" photosDecoder
    |> required "subfolders" folderDecoder
```

Just like in the case of `str = List.reverse str`, we're trying to define a value in terms of a value we haven't finished defining yet! This definition will also expand forever.

FIXING THE CYCLIC DEFINITION USING DECODE.LAZY

We can solve this problem using `Decode.lazy`. It has this type:

```
Decode.lazy : (() -> Decoder val) -> Decoder val
```

NOTE We first saw the `()` type in Chapter 6. It's called *Unit*, and it holds no information.

We can use `Decode.lazy` to remove the cyclic definition from `folderDecoder` like so:

```
folderDecoder =
```

```
Decode.succeed folderFromJson
  |> required "name" string
  |> required "photos" photosDecoder
  |> required "subfolders" (Decode.lazy (\_ -> list folderDecoder))
```

NOTE The `list` here can go on either side of the `Decode.lazy`; we could also have written this as `(list (Decode.lazy (_ -> folderDecoder)))`. While that way also works, it adds more visual separation between `list` and `folderDecoder`, making it harder to see which decoder we'll use for "subfolders".

Here we no longer have the neverending expansion problem, because once the compiler reaches `(_ -> list folderDecoder)` it stops expanding. That expression is already a fully-formed anonymous function, which needs no further expansion!

Sure, later on when this decoder actually gets run on some real JSON, `Decode.lazy` will call this anonymous function to obtain `folderDecoder`...but that will work just fine! By that time, `folderDecoder` will already have been successfully defined. This is the situation `Decode.lazy` was created to solve, and it will do the trick nicely here.

TIP If you see this `BAD RECURSION` error on a decoder, it's likely that `Decode.lazy` can resolve it.

Now that we have a working decoder for `root : Folder`, we'll turn to the last decoder our model requires: a decoder for its `photos : Dict String Photo` field, the single source of truth for our detailed photo data.

7.3.3 Accumulating while Decoding

We already have a `Decoder Folder` that works on this JSON structure, but it's perfectly reasonable for us to create a second decoder that also works on the same JSON!

We'll do that when we create the final `Photo` dictionary we'll use in our `Model`. Add this to the end of your `PhotoFolders.elm` file:

```
modelPhotosDecoder : Decoder (Dict String Photo)
modelPhotosDecoder =
  Decode.succeed modelPhotosFromJson
    |> required "photos" photosDecoder
    |> required "subfolders" (Decode.lazy (\_ -> list modelPhotosDecoder))

modelPhotosFromJson :
  Dict String Photo
  -> List (Dict String Photo)
  -> Dict String Photo
modelPhotosFromJson folderPhotos subfolderPhotos =
  List.foldl Dict.union folderPhotos subfolderPhotos
```

NOTE `modelPhotosDecoder` doesn't bother to decode the "name" field because it never uses the folders' names, only their photos.

This new `modelPhotosDecoder` will traverse the same JSON structure as `folderDecoder`, which is why they have so many similarities:

- Both use `required "photos" photosDecoder`
- Both use `Decode.lazy with list` to recursively decode the "subfolders" field
- Both give `Decode.succeed` a handwritten function instead of a record constructor

We see some new functions in `photosFromJson`, namely `Dict.union` and `List.foldl`. What do they do?

DICT.UNION

Let's start with `Dict.union`. Here is its type:

```
union : Dict comparable val -> Dict comparable val -> Dict comparable val
```

It iterates over the first dictionary and calls `Dict.insert` on each of its keys and values, inserting them into the second dictionary. The returned dictionary has the combined contents of both dictionaries!

NOTE Because the calls to `Dict.insert` use keys and values from the first dictionary, anytime the second dictionary already happened to have an entry for a particular key, it will get overridden.

We can use this to combine our various `Photo` dictionaries into the single dictionary `Model` needs. However, since `Dict.union` only combines two individual dictionaries—while we need to combine many dictionaries—`Dict.union` will need some help.

That's where `List.foldl` comes in!

LIST.FOLDL AND LIST.FOLDR

The `List` module offers two *fold* functions: `List.foldl` and `List.foldr`. ("foldl" is short for "fold from the left" and "foldr" is short for "fold from the right." JavaScript calls these functions "reduce" and "reduceRight" because they reduce collections down to a single value. Any naming similarity between the `foldr` function and our `Folder` type is purely coincidental!)

We can compare what `foldl` and `foldr` do by calling both functions in `elm-repl`, passing the same arguments in each case:

```
> List.foldl (\letter str -> str ++ "-" ++ String.fromChar letter) "start" [ 'a', 'b', 'c', 'd' ]
"start-a-b-c-d" : String

> List.foldr (\letter str -> str ++ "-" ++ String.fromChar letter) "start" [ 'a', 'b', 'c', 'd' ]
"start-d-c-b-a" : String
```

Let's walk through what's happening here.

NAMING THE INGREDIENTS

First, note that both `List.foldl` and `List.foldr` have the same type:

```
(element -> state -> state) -> state -> List element -> state
```

We'll give those arguments some names:

1. `(element -> state -> state)` is our *update function*.
2. `state` is our *initial state*.
3. `List element` is our *list*.

We've named the first argument the *update function* because it works similarly to the Elm Architecture's `update` function, specifically the version we used in Chapter 2:

```
Elm Architecture update:      Msg -> Model -> Model
Fold function update:        element -> state -> state
```

Whenever the Elm Runtime calls `update`, it passes a `Msg` and the current `Model`, and gets back the updated `Model` it will use the next time it calls `update`.

Similarly, each time a fold calls its *update function*, it passes an `element` and the previous `state`, and gets back the updated `state` to use the next time it calls the *update function*.

The first time the fold function calls its *update function*, it passes the *initial state* as the `state` argument and the first element in the *list* as the other argument. After repeating this process with the remaining elements in the *list*, it returns the final `state` value. (If the list is empty, it returns the *initial state* immediately.)

DIFFERENCES BETWEEN FOLDL AND FOLDR

Although `foldl` and `foldr` both call their *update functions* on each element in the list, they differ in the order in which they pass those elements to their *update functions*.

Table 7.5 compares the calls `foldl` and `foldr` make for the same arguments.

Table 7.5 Calling `foldl` and `foldr` on the same arguments

List.foldl update ['a', 'b', 'c', 'd'] "start"	List.foldr update ['a', 'b', 'c', 'd'] "start"
"start"	"start"
> update 'a'	> update 'd'
> update 'b'	> update 'c'
> update 'c'	> update 'b'
> update 'd'	> update 'a'

Both `foldl` and `foldr` begin with the *initial state* and then call the *update function* on it four times. The difference is that where `foldl` passes each element in its list to the *update function* in the same order as how they appear in that list, `foldr` passes them in the reverse order.

USING LIST.FOLDL WITH DICT.UNION

Now that we know how `foldl` works, let's break down the types involved in the expression where we've called it.

Table 7.9 Types involved in `(List.foldl Dict.union)` plus two more arguments from Decoder outputs

Value	Type
<code>List.foldl</code>	<code>(element -> state -> state) -> state -> List element -> state</code>
<code>Dict.union</code>	<code>Dict comparable val -> Dict comparable val -> Dict comparable val</code>
<code>folderPhotos</code>	<code>Dict String Photo</code>
<code>subfolderPhotos</code>	<code>List (Dict String Photo)</code>

In this expression, `folderPhotos` is our start value, so `List.foldl` will perform these steps:

1. Pass `Dict.union` the first dictionary in the `subfolderPhotos` list, with `folderPhotos` as the second argument. The result is our first `state` value.
2. Call `Dict.union` passing the next dictionary in the `subfolderPhotos` list, with the previous outcome as the second argument. The result is our new `state` value.
3. Repeat the previous steps until we've iterated through the entire `subfolderPhotos` list. `List.foldl` returns the final `state` value.

In this way, we end up with a `Dict String Photo` value that combines all the photo information from the current folder as well as its subfolders!

NOTE The only time `List.foldl Dict.union` and `List.foldr Dict.union` will produce different values is if there are any duplicate keys in the the dictionaries involved. If that happens, the choice of `foldl` or `foldr` will change which values get overridden. We don't expect any duplicate keys, and even if we did, we wouldn't care how they got overridden, so we're choosing `foldl` only because it runs faster than `foldr`.

JOINING TWO DECODERS

Finally, it's time to for our `folderDecoder` and `modelPhotosDecoder` to team up and form a working `modelDecoder`! Let's replace the definition of `modelDecoder` with this:

```
modelDecoder : Decoder Model
modelDecoder =
  Decode.map2
    (\photos root ->
      { photos = photos, root = root, selectedPhotoUrl = Nothing }
    )
    modelPhotosDecoder
    folderDecoder
```

Like its smaller cousin `map`, the `Decode.map2` function transforms the *contents* of a value—in this case, a `Decoder` value—rather than the entire value. The difference is that `map2` takes an extra argument. Here are their types side by side:

```
map : (val -> final) -> Decoder val -> Decoder final
```

```
map2 : (val1 -> val2 -> final) -> Decoder val1 -> Decoder val2 -> Decoder final
```

We used `map2` instead of `map` because building our `Model` requires the outputs of both decoders. We also could have used `Decode.andThen` for this purpose, which works similarly to the `Maybe.andThen` function we saw in Section 7.1.2, but it's generally better to use the simpler function (in this case, `map2`) when either of two functions would do the job. (In practice, `Decode.andThen` tends to be used to perform validation on freshly decoded values.)

TIP Many other modules like `Result` and `Random` offer similar `map2` and `andThen` functions.

With these two decoders' powers combined, our `modelDecoder` is now fully operational!

NOTE If you're looking for extra challenges, you can try combining `modelPhotosDecoder` and `folderDecoder` into a single `Decoder (Folder, Dict String Photo)` that decodes both the folders and the photos in one pass! That version would use `Decode.map` instead of `Decode.map2` here.

VIEWING THE FINISHED PAGE

After we recompile, the page should now load both photos and folders from the server. The result should be more folders and more photos than we previously had hardcoded!

The final page should look something like this, depending on which photo you select:

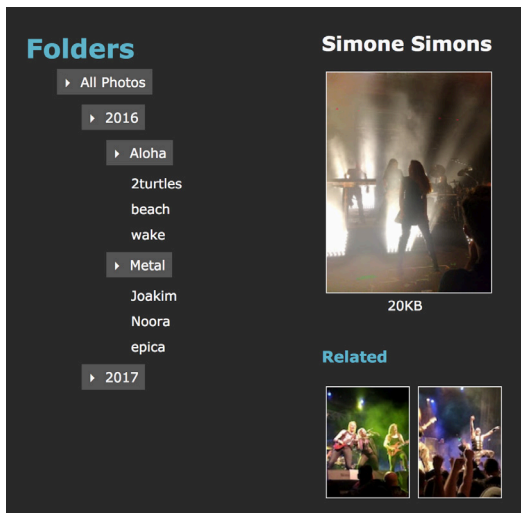


Figure 7.7 The final Photo Folders page

Now the Photo Folders page is loading its model data from the server. It gathers all the photos it finds in the folder JSON into a single `Dict String Photo`, which the rest of our page uses to make the folders and photos interactive.

We did it!

7.4 Summary

In this chapter we built a whole new page from scratch! The Photo Folders feature is sure to be a hit, thanks to our efforts. Along the way we learned several new techniques, including:

- Using `Dict String Photo` to store photos by their URL.
- Using a recursive union type of type `Folder = Folder { subfolders : List Folder, ... }` to represent a tree of folders with subfolders nested inside.
- Using a recursive `Msg` constructor to implement expanding and collapsing on that tree.
- Using `Decode.keyValuePairs` to decode an object into key-value pairs for further processing with functions like `Decode.map`.
- Creating intermediate representations like `Decoder JsonPhoto` to decode what we can from a particular JSON object as we can, then using `Decode.map` to use additional information to translate from `Decoder JsonPhoto` into a `Decoder Photo`. We used this technique to decode a photo's URL from its key in the enclosing JSON object.
- Using recursive decoders to decode tree structures, and how `Decode.lazy` can fix cyclic definition errors in recursive decoders.
- Using `Dict.union` to combine the contents of two dictionaries into one.
- Using `List.foldl` to reduce a list of values down to one value.
- Using the `map2` function to combine and transform two different values—in this case, JSON decoders—into one value.

Here is the final `PhotoFolders.elm` file.

Listing 7.11 PhotoFolders.elm

```
module PhotoFolders exposing (main)

import Dict exposing (Dict)
import Html exposing (..)
import Html.Attributes exposing (class, src)
import Html.Events exposing (onClick)
import Http
import Json.Decode as Decode exposing (Decoder, int, list, string)
import Json.Decode.Pipeline exposing (required)

type Folder
  = Folder
    { name : String
    , photoUrls : List String
    , subfolders : List Folder
    , expanded : Bool
    }
```

```

type alias Model =
  { selectedPhotoUrl : Maybe String
  , photos : Dict String Photo
  , root : Folder
  }

initialModel : Model
initialModel =
  { selectedPhotoUrl = Nothing
  , photos = Dict.empty
  , root = Folder { name = "Loading...", expanded = True, photoUrls = [], subfolders = [] }
  }

init : ( Model, Cmd Msg )
init =
  ( initialModel
  , modelDecoder
    |> Http.get "http://elm-in-action.com/folders/list"
    |> Http.send LoadPage
  )

modelDecoder : Decoder Model
modelDecoder =
  Decode.map2
    (\photos root ->
      { photos = photos, root = root, selectedPhotoUrl = Nothing }
    )
    modelPhotosDecoder
    folderDecoder

type Msg
  = SelectPhotoUrl String
  | LoadPage (Result Http.Error Model)
  | ToggleExpanded FolderPath

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    ToggleExpanded path ->
      ( { model | root = toggleExpanded path model.root }, Cmd.none )

    SelectPhotoUrl url ->
      ( { model | selectedPhotoUrl = Just url }, Cmd.none )

    LoadPage (Ok newModel) ->
      ( newModel, Cmd.none )

    LoadPage (Err _) ->
      ( model, Cmd.none )

```

```

view : Model -> Html Msg
view model =
  let
    photoByUrl : String -> Maybe Photo
    photoByUrl url =
      Dict.get url model.photos

    selectedPhoto : Html Msg
    selectedPhoto =
      case Maybe.andThen photoByUrl model.selectedPhotoUrl of
        Just photo ->
          viewSelectedPhoto photo

        Nothing ->
          text ""
  in
  div [ class "content" ]
    [ div [ class "folders" ]
      [ h1 [] [ text "Folders" ]
        , viewFolder End model.root
      ]
      , div [ class "selected-photo" ] [ selectedPhoto ]
    ]

main : Program Never Model Msg
main =
  Html.program { init = init, view = view, update = update, subscriptions = \_ -> Sub.none
    }

type alias Photo =
  { title : String
  , size : Int
  , relatedUrls : List String
  , url : String
  }

viewPhoto : String -> Html Msg
viewPhoto url =
  div [ class "photo", onClick (SelectPhotoUrl url) ]
    [ text url ]

viewSelectedPhoto : Photo -> Html Msg
viewSelectedPhoto photo =
  div
    [ class "selected-photo" ]
    [ h2 [] [ text photo.title ]
      , img [ src (urlPrefix ++ "photos/" ++ photo.url ++ "/full") ] []
      , span [] [ text (toString photo.size ++ "KB") ]
      , h3 [] [ text "Related" ]
      , div [ class "related-photos" ]
        (List.map viewRelatedPhoto photo.relatedUrls)
    ]

```

```

viewRelatedPhoto : String -> Html Msg
viewRelatedPhoto url =
  img
    [ class "related-photo"
    , onClick (SelectPhotoUrl url)
    , src (urlPrefix ++ "photos/" ++ url ++ "/thumb")
    ]
  []

viewFolder : FolderPath -> Folder -> Html Msg
viewFolder path (Folder folder) =
  let
    viewSubfolder : Int -> Folder -> Html Msg
    viewSubfolder index subfolder =
      viewFolder (appendIndex index path) subfolder

    folderLabel =
      label [ onClick (ToggleExpanded path) ] [ text folder.name ]
  in
  if folder.expanded then
    let
      contents =
        List.append
          (List.indexedMap viewSubfolder folder.subfolders)
          (List.map viewPhoto folder.photoUrls)
    in
    div [ class "folder expanded" ]
      [ folderLabel
      , div [ class "contents" ] contents
      ]
  else
    div [ class "folder collapsed" ] [ folderLabel ]

appendIndex : Int -> FolderPath -> FolderPath
appendIndex index path =
  case path of
    End ->
      Subfolder index End

    Subfolder subfolderIndex remainingPath ->
      Subfolder subfolderIndex (appendIndex index remainingPath)

urlPrefix : String
urlPrefix =
  "http://elm-in-action.com/"

type FolderPath
  = End
  | Subfolder Int FolderPath

toggleExpanded : FolderPath -> Folder -> Folder
toggleExpanded path (Folder root) =

```

```

case path of
  End ->
    Folder { root | expanded = not root.expanded }

  Subfolder targetIndex remainingPath ->
    let
      subfolders : List Folder
      subfolders =
        List.indexedMap transform root.subfolders

      transform : Int -> Folder -> Folder
      transform currentIndex currentSubfolder =
        if currentIndex == targetIndex then
          toggleExpanded remainingPath currentSubfolder
        else
          currentSubfolder
    in
      Folder { root | subfolders = subfolders }

type alias JsonPhoto =
{ title : String
, size : Int
, relatedUrls : List String
}

jsonPhotoDecoder : Decoder JsonPhoto
jsonPhotoDecoder =
  Decode.succeed JsonPhoto
    |> required "title" string
    |> required "size" int
    |> required "related_photos" (list string)

finishPhoto : ( String, JsonPhoto ) -> ( String, Photo )
finishPhoto ( url, json ) =
  ( url
  , { url = url
    , size = json.size
    , title = json.title
    , relatedUrls = json.relatedUrls
    }
  )

fromPairs : List ( String, JsonPhoto ) -> Dict String Photo
fromPairs pairs =
  pairs
    |> List.map finishPhoto
    |> Dict.fromList

photosDecoder : Decoder (Dict String Photo)
photosDecoder =
  Decode.keyValuePairs jsonPhotoDecoder
    |> Decode.map fromPairs

```



```

folderDecoder : Decoder Folder
folderDecoder =
    Decode.succeed folderFromJson
        |> required "name" string
        |> required "photos" photosDecoder
        |> required "subfolders" (Decode.lazy (\_ -> list folderDecoder))

folderFromJson : String -> Dict String Photo -> List Folder -> Folder
folderFromJson name photos subfolders =
    Folder
        { name = name
        , expanded = True
        , subfolders = subfolders
        , photoUrls = Dict.keys photos
        }

modelPhotosDecoder : Decoder (Dict String Photo)
modelPhotosDecoder =
    Decode.succeed modelPhotosFromJson
        |> required "photos" photosDecoder
        |> required "subfolders" (Decode.lazy (\_ -> list modelPhotosDecoder))

modelPhotosFromJson :
    Dict String Photo
    -> List (Dict String Photo)
    -> Dict String Photo
modelPhotosFromJson folderPhotos subfolderPhotos =
    List.foldl Dict.union folderPhotos subfolderPhotos

```

A

Getting Set Up

This covers installing

- Node.js (6.9.2 or higher) and NPM
- The Elm Platform
- The Elm in Action Repository
- Recommended optional tools

INSTALLING NODE.JS AND NPM

In addition to having the Elm Platform installed, the examples in this book require having Node.js 6.9.2 or higher and NPM 3.10 or higher installed as well. If you haven't already, visit <https://nodejs.org> and follow the instructions to download and install them.

NOTE The Node.js installer on <https://nodejs.org> also installs NPM for you, so if you install Node that way, you will not need to install NPM separately.

To confirm that you have Node.js 6.9.2 or higher and NPM 3.10 or higher installed, you should be able to run these commands in your terminal and see similar output:

```
$ node --version
v6.9.2

$ npm --version
3.10.9
```

If you have Node installed but not NPM, then your installation of Node is probably not the one from <http://nodejs.org>. Please make sure you have NPM installed before continuing!

INSTALLING THE ELM PLATFORM

Now that you have NPM installed, you can use it to get the Elm Platform along with the `elm-test` and `elm-css` command line interfaces:

```
npm install -g elm elm-test elm-css
```

TIP If `npm` gives you a lengthy error involving the word `EACCESS`, visit <https://docs.npmjs.com/getting-started/fixing-npm-permissions> for how to resolve it. If you are unable to resolve it, you can fall back on installing the Elm Platform directly from <http://elm-lang.org/install> - but this will only get you through Chapters 1 through 5. Starting in Chapter 6, being able to run `npm install -g` will be required to run the examples!

Let's verify that the Elm platform was installed properly:

```
elm-make --version
elm-make 0.18.0 (Elm Platform 0.18.0)
```

If you see a version higher than `0.18.0`, then as of this writing, you are living in the future! Hello from the past, where `0.18.0` is the latest release.

NOTE Any version that starts with `0.18` should work fine with this book, but according to semantic versioning, a version number beginning with `0.19` or higher indicates breaking changes. In such a case, there's a good chance your examples will not compile! Try `npm install -g elm@0.18.0` to get a compatible version.

OBTAINING THE ELM IN ACTION REPOSITORY

This book does not require using Git or any Git knowledge. If you are not a Git user, you can download and extract a zip archive of the repository at <https://github.com/rtfeldman/elm-in-action/archive/master.zip> and proceed directly to the next section!

Git users can download the repository by running this:

```
git clone https://github.com/rtfeldman/elm-in-action.git
```

The repository has been tagged with various checkpoints. Suppose you are about to read Chapter 2, Section 2.2.1. You can visit this URL to obtain all the code necessary to work through the examples in Section 2.2.1:

```
https://github.com/rtfeldman/elm-in-action/tree/2.2.1
```

You can replace the `2.2.1` in that URL with a different section number to view the code at the start of that section.

For example, if you want to peek ahead and see where things stand at the end of Section 2.2.1, you can bring up the `2.2.2` tag by visiting <https://github.com/rtfeldman/elm-in-action/tree/2.2.2> in a browser. Alternatively, you can run `git checkout 2.2.2` from a terminal if you ran `git clone` on the repository earlier.

INSTALLING RECOMMENDED OPTIONAL TOOLS

To get syntax highlighting and other niceties, visit <http://elm-lang.org/install#syntax-highlighting> and select your editor of choice to find an appropriate Elm plugin. Make sure your editor is configured to convert tabs to spaces, as tab characters are syntax errors in Elm!

I also strongly recommend installing *elm-format*: <https://github.com/avh4/elm-format>

I absolutely love *elm-format*. It automatically formats Elm code according to a nice, consistent standard. I use it for personal projects, my whole team at work uses it, and I have my editor configured to run it whenever I save. (Several editors enable this by default, but if not, there are easy instructions in the above link for how to enable format-on-save.)

All of the examples in this book are formatted with *elm-format*, so using it will make life easier as you work through them!