Elm in Action

Richard Feldman

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Elm in Action
Version 1

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

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 refactor, 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 got 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 crazy 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 single 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, using its vast library ecosystem to avoid reinventing the wheel. 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.

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.

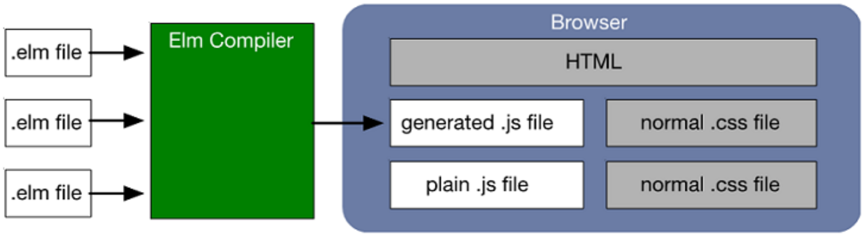

Figure 1.1 Elm files are compiled to plain old JavaScript

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 maintainablity 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 a simpler language than JavaScript, it's also much younger. The ecosystem is smaller, which means you'll encounter fewer problems where someone has already published an off-the-shelf solution. Elm code can interoperate with JavaScript code to piggyback that 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'd stick to a more familiar language and toolset for projects like these:

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

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-repl 0.17.1 -----------------------------------------------------
 :help for help, :exit to exit, more at https://github.com/elm-lang/elm-repl
--------------------------------------------------------------------------
>
```

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 " ++ toString pi ++ " (give or take)"   ❶
"Pi is 3.141592653589793 (give or take)"
```

❶ `toString` **is a standalone function, not a method**

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) ❶
90210

> 2 ^ 11
```

```
2048

> 49 / 10
4.9

> 49 // 10   ❷
4

> -5 % 2   ❸
1
```

❶ Nest expressions via parentheses
❷ Integer division (decimals get truncated)
❸ 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

Comments also work a bit differently in Elm:

- Use `--` instead of `//` for inline comments
- Use `{- -}` instead of `/* */` for block comments

Let's see these in action!

Listing 1.3 Characters, Comments, and Constants

```
> 'a'  -- This is a comment. It will be ignored.  ❶
'a'

> "a"  {- This is a block comment. It will also be ignored. -}  ❷
"a"

> milesPerHour = 88  ❸
88

> milesPerHour
88
```

❶ **JavaScript comment: //**
❷ **JavaScript comment: /* ... */**
❸ **JavaScript: const milesPerHour = 88;**

NAMING VALUES WITH CONSTANTS

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

> **DEFINITION** **A *constant* assigns a name to a value. Once assigned, this name cannot be later reassigned to a different value in the same scope.**

There are a few things to keep in mind when naming constants.

- The name must begin with a lowercase letter. After that it can be a mix of letters, numbers, underscores, and apostrophes.
- By convention, all letters should be in one uninterrupted sequence. For example, `map4` is a reasonable name, but `map4ever` is not, because 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 constants. Use `camelCase` instead.
- It can be tempting to name a new constant by recycling an old name and slapping an apostrophe or underscore on the end, like defining `list` and then defining `list'` or `list_` shortly after. This is a code smell which leads to similar readability problems as undescriptive single-letter identifiers do in JavaScript. Take the time to pick a more descriptive name; whoever reads your code next will be glad you did!
- If you absolutely must know whether the compiler will accept `richard's_rad__TH1NG` as a valid constant name, remember: what happens in `elm-repl` stays in `elm-repl`.

ASSIGNING CONSTANTS TO EXPRESSIONS

Not only can you assign constants 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 lots of smaller ones!

1.2.3 Booleans and Conditionals

There aren't terribly 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 insetad of JavaScript's `!` prefix

Let's try them out!

Listing 1.4 Boolean Expressions

```
> pi == pi   ❶
True   ❷

> pi /= pi   ❸
False   ❹

> not (pi == pi)   ❺
False

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

> 3 < pi && pi < 4 ❻
True
```

❶ **JavaScript:** `pi === pi`
❷ **JavaScript:** `true`
❸ **JavaScript:** `pi !== pi`
❹ **JavaScript:** `false`
❺ **JavaScript:** `!(pi === pi)`
❻ `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.

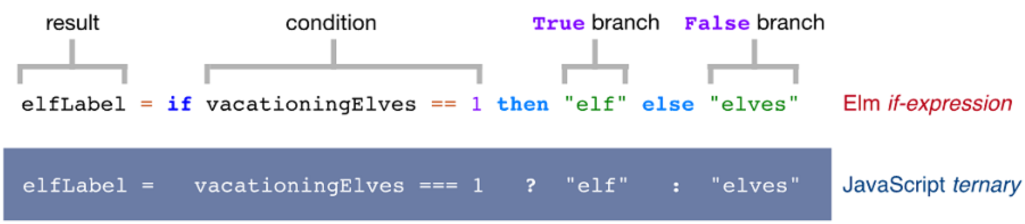

Figure 1.2 Comparing an Elm *if-expression* to a JavaScript *ternary*

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

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

Each of these ingredients must be expressions, and the *if-expression* itself 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                                  ❶
  "elf"                                                ❶
else                                                   ❶
  (if elfCount >= 0 then "elves" else "anti-elves")    ❶


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


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

❶ **Use an *if-expression* inside `else`**
❷ **Rearrange some whitespace**
❸ **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.

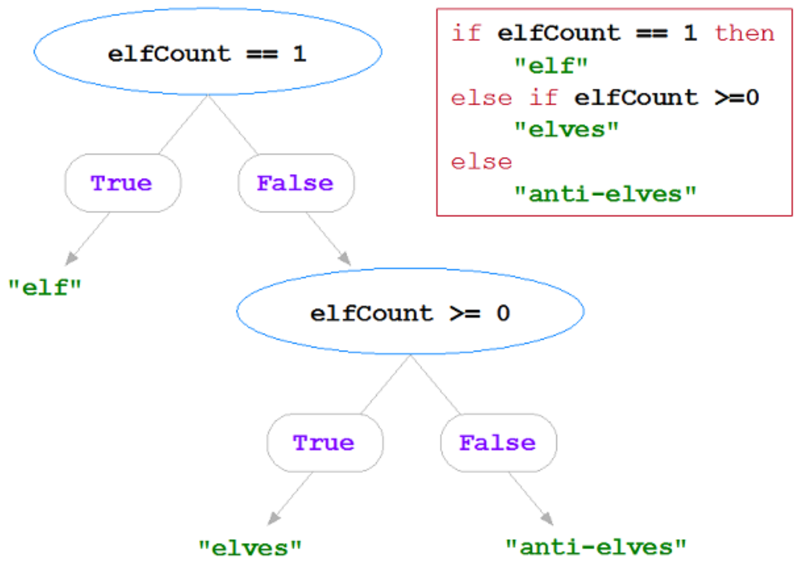

Figure 1.3 The *else if* technique: use an *if-expression* as the *else* branch of another *if-expression*

Nesting expressions for fun and profit is a recurring theme in Elm, and we'll see plenty more recipes like `else if` thoughout 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: `isOdd`. It will take a number and then:

- Return `True` if the number is odd
- Return `False` if the number is even

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

Listing 1.7 Defining a function

```
> isOdd num = num % 2 == 1    ❶
<function>

> isOdd 5   ❷
True

> (isOdd 5) ❸
True

> isOdd (5 + 1) ❹
False
```

❶ JavaScript: function isOdd(num) { return num % 2 === 1; }
❷ JavaScript: isOdd(5)
❸ JavaScript: (isOdd(5))
❹ JavaScript: isOdd(5 + 1)

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 `isOdd` function, the expression `num % 2 == 1` 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;    ❶
    }

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

❶ **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 semantically analogous 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    ❶
<function>

> pluralize "elf" "elves" 3  ❷
"elves"

> pluralize "elf" "elves" (round 0.9)  ❸
"elf"
```

❶ Don't forget to indent!
❷ No commas between arguments!
❸ (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 toolbox 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 two of its functions, `toLower` and `toUpper`.

Listing 1.9 Importing functions

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

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

Observant readers may note a striking resemblance between Elm's `String.toUpper` 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 istelf.

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 `toLower` and `toUpper`.

Table 1.2 String Functionality Comparison

JavaScript	Elm
`"storm".length`	`String.length "storm"`
`"dredge".toUpperCase()`	`String.toUpper "dredge"`
`String.fromCharCode(something)`	`String.fromChar something`

Not only is this organizational pattern consistent within the `String` module, it's consistent across Elm. Want a standard date-related feature? Look no further than the functions in the `Date` module. Regular Expression functions? Hit up the `Regex` module.

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

TIP **Complete documentation for `String`, `Date`, `Regex`, 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 must give `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 telephone number.

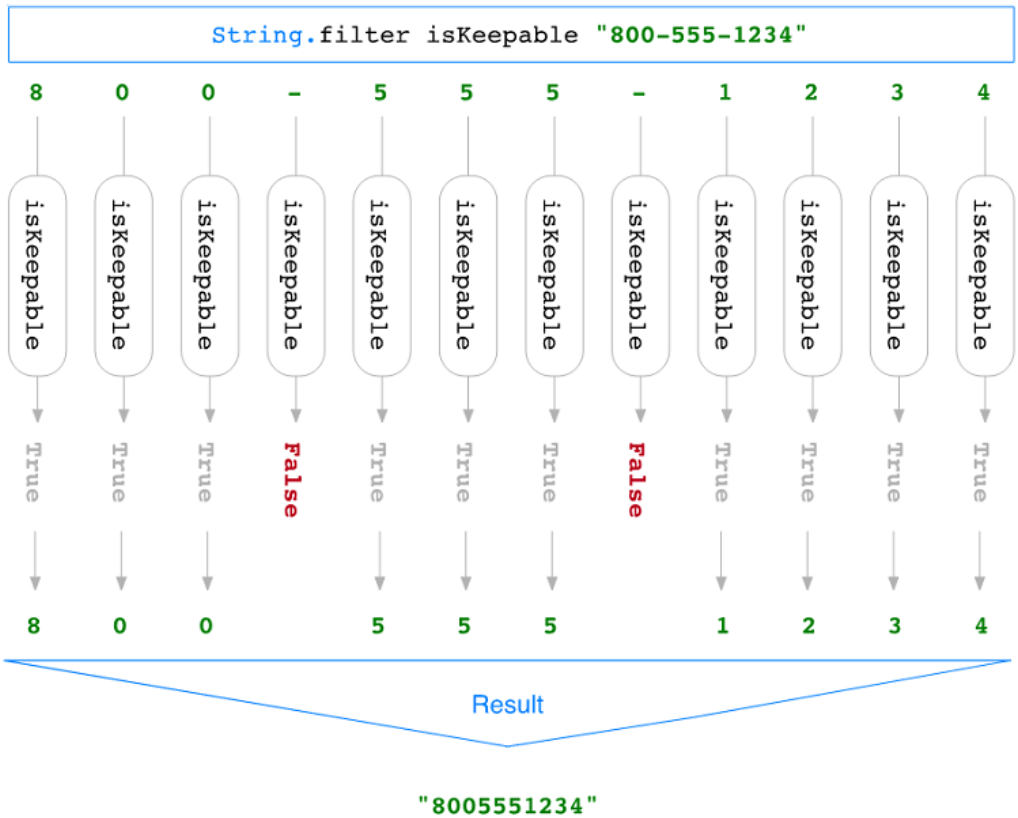

Figure 1.4 Using String.filter to remove dashes from a phone number

As in JavaScript, 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

```
> import String
> isKeepable character = character /= '-'   ❶
<function>

> isKeepable 'z'
True

> isKeepable '-'
False

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

❶ **A function describing which characters to keep**
❷ **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 constants to an expression.**

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

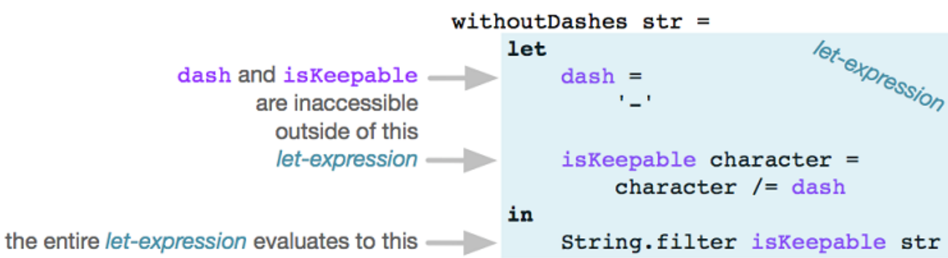

Figure 1.5 Anatomy of the wild *let-expression*

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

```
> import String
> 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*.

Whenever you see a *let-expression*, you can mentally replace it with the part after its `in` keyword—in this case, `String.filter isKeepable str`. All the values between `let` and `in` are intermediate constants 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 constants inside function bodies, branches of *if-expressions*, or anyplace else.

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

1.3.4 Anonymous Functions

Anonymous functions in Elm work the same way they do in JavaScript: like named functions but without the name.

Listing 1.11 Named and Anonymous Functions

```
function area(w, h) { return w * h; }  ❶

function(w, h) { return w * h; }  ❷

area w h = w * h  ❸

\w h -> w * h  ❹
```

❶ **JavaScript named function**
❷ **JavaScript anonymous function**
❸ **Elm named function**
❹ **Elm anonymous function**

Elm's anonymous functions differ from its 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

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

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

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

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

❶ **Our simple filter fell short here**
❷ **Much better!**
❸ **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!

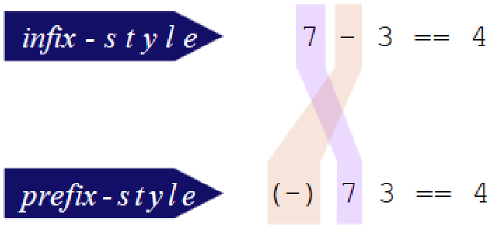

Figure 1.6 Calling the (-) operator in both *infix-style* and *prefix-style*

Let's play with some operators in `elm-repl`:

Listing 1.13 Operators are functions

```
> (/)
<function>

> divideBy = (/)
<function>

> 7 / 2      ❶
3.5

> (/) 7 2      ❷
3.5

> divideBy 7 2
3.5

> 7 `divideBy` 2      ❸
3.5
```

❶ *infix-style* calling
❷ *prefix-style* calling
❸ backticks call normal functions in *infix-style*

As Listing 3.4 demonstrates, you can also call normal functions in *infix-style* by surrounding them with backticks.

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, you see `+` first, then `==`, and finally `-`. In the *prefix-style* expression, the order is different: first you see `==`, then `+`, and finally `-`. Why is this?

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

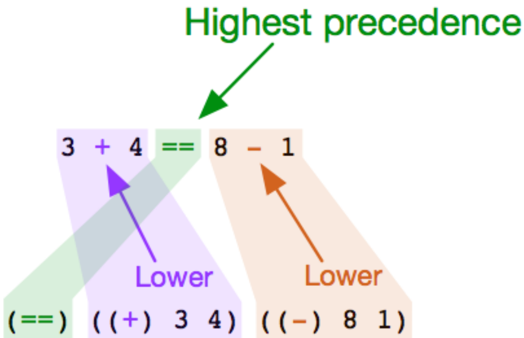

Figure 1.7 Higher precedence means (==) gets evaluated before (+) and (-).

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.

We'll dig into operator precedence in greater depth in later chapters.

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 normal function calls have higher precedence than any operator. This means any time you want to pass the results of two normal 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* or *right-associative*. Every operator is one or the other.

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	`10 - 6 - 3`	`1`
assuming *left-associative*	`((10 - 6) - 3)`	`1`
assuming *right-associative*	`(10 - (6 - 3))`	`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! We'll use a mix of left-associative and right-associative operators throughout the book.

In Chapter 7, we'll see how to define our own operators, including specifying their associativity and precedence values.

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.)

- You can combine it with another list using the `++` operator. In JavaScript this is done with the `concat` method rather than an operator.
- 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 *union types* that we'll cover in Chapter 3.)

Although Elm supports both (immutable) lists and (also immutable) arrays, lists are used far more often because they have better performance characteristics in typical Elm use cases. We'll get more into the performance differences between linked lists and arrays in Chapter 10.

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
`[ 1, 2, 3 ].length`	`List.length [ 1, 2, 3 ]`
`[ "one fish", "two fish" ][0]`	`List.head [ "one fish", "two fish" ]`
`[ "one fish", "two fish" ][1]`	**No arbitrary position-based element access**
`[ 1, 2 ].concat([ 3, 4 ])`	`[ 1, 2 ] ++ [ 3, 4 ]`
`[ 1, 2 ].push(3)`	**Cannot be modified; use e.g.** `append` **instead**
`[ 1, "Paper", 3 ]`	**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 consider the `List.filter` function, which works like the `String.filter` function we used earlier.

Recall 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`!

> **TIP** Whereas before we had to `import String`, the `List` module is one of a select few modules that gets imported by default. We don't need to write `import List` to reference it.

Listing 1.14 Filtering lists

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

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

```
["ZZ","Top"]

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

> List.filter (\num -> num % 2 == 1) [ 1, 2, 3, 4, 5 ]  ❹
[1,3,5]
```

❶ **Same function we passed to String.filter earlier**
❷ **Strings instead of characters**
❸ **Works just like with String.filter**
❹ **Keep only the odd numbers**

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

```
[ 1, 2, 3, 4, 5 ].filter(function(num) { return num % 2 === 1; })
```

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?

```
[ 1, "2", "cat", 4, "5", "" ].filter(function(num) { return num % 2 === 1; })
```

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 successfully returns `[ 1, "5" ]`. (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
`{ name: "Li", cats: 2 }`	`{ name = "Li", cats = 2 }`
`({ name: "Li", cats: 2 }).toString()`	`toString { name = "Li", cats = 2 }`
`({ name: "Li", cats: 2 }).cats`	`({ name = "Li", cats = 2 }).cats`
`({ name: "Li", cats: 2 })["cats"]`	**Fields can only be accessed directly, using a dot**

`({ name: "Li", cats: 2 }).cats = 3`	**Cannot be modified. (New cat? New record!)**
`{ NAME: "Li", CATS: 2 }`	**Fields have the same naming rules as constants**
`({ name: "Li", cats: 2 }).__proto__`	**No prepackaged fields, only the ones you define**
`Object.keys({ name: "Li", cats: 5 })`	**No listing of field names is available on demand**
`Object.prototype`	**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. (As we will see in Chapter 10, behind the scenes Elm does not actually copy the entire record—that would be slow!—but rather only the parts that will be different.)

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 }  ❶
{ name = "Li", cats = 3 }

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

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

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

❶ **Record update syntax**
❷ **Original record unmodified!**
❸ **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)

> fst ( "Tech", 9 )   ❶
"Tech"

> snd ( "Tech", 9 )   ❷
9
```

❶ Return first element (only works on 2-element tuples)
❷ Return second element (only works on 2-element tuples)

You can only use the `fst` and `snd` functions on tuples that contain two elements. If they have more than two, you can use *tuple destructuring* to extract their values.

DEFINITION *Tuple destructuring* extracts the values inside a tuple and assigns them to constants 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  ❶
<function>

> multiply3d ( 6, 7, 2 )
84

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

❶ Destructuring a tuple into three constants: x, y, and z
❷ 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 constant.

> **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 tuples can be represented (more verbosely) using a record instead, it's often better to refactor long tuples—say, with more than three elements—into records. 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 and lists, whereas the `+` operator is for addition only.
- Double quotes refer to strings. Single quotes refer to individual UTF-8 characters.
- *let-expressions* introduce scoped constants 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, and their elements must share a consistent type.
- `List.filter (\num -> num < 0) numbersList` returns a list containing all the negative numbers in the original `numbersList`
- `catLover = { name = "Li", cats = 2 }` assigns a record to the constant `catLover`. Once assigned, constants 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 )` destructures the first two fields of a tuple such as `( 2, 4, 6, 8 )`. In this example, `foo` would be `2` and `bar` would be `4`.

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

Table 1.6 Differences between Elm and JavaScript

Elm	JavaScript
`-- This is an inline comment`	`// This is an inline comment`
`{- This is a block comment -}`	`/* This is a block comment */`
`True && False`	`true && false`
`"Ahoy, " ++ "World!"`	`"Ahoy, " + "World!"`
`"A spade" == "A spade"`	`"A spade" === "A spade"`
`"Calvin" /= "Hobbes"`	`"Calvin" !== "Hobbes"`
`2 ^ 11`	`Math.pow(2, 11)`
`-49 // 10`	`Math.trunc(-49 / 10)`
`if n % 2 == 1 then "odd" else "even"`	`n % 2 === 1 ? "odd" : "even"`
`List.filter (\n -> n % 2 == 1) nums`	`nums.filter(function(n) { ... })`
`pluralize singular plural count = ...`	`function pluralize(s, p, c) { ... }`

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

Table 1.7 Differences between normal functions and operators

Function	How to identify one	Calling style	Examples
Normal	**Name begins with a letter**	***prefix-style***	`negate`, `not`, `toString`
Operator	**Name has no letters or numbers**	***infix-style***	`(++)`, `(*)`, `(==)`

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...erm...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.

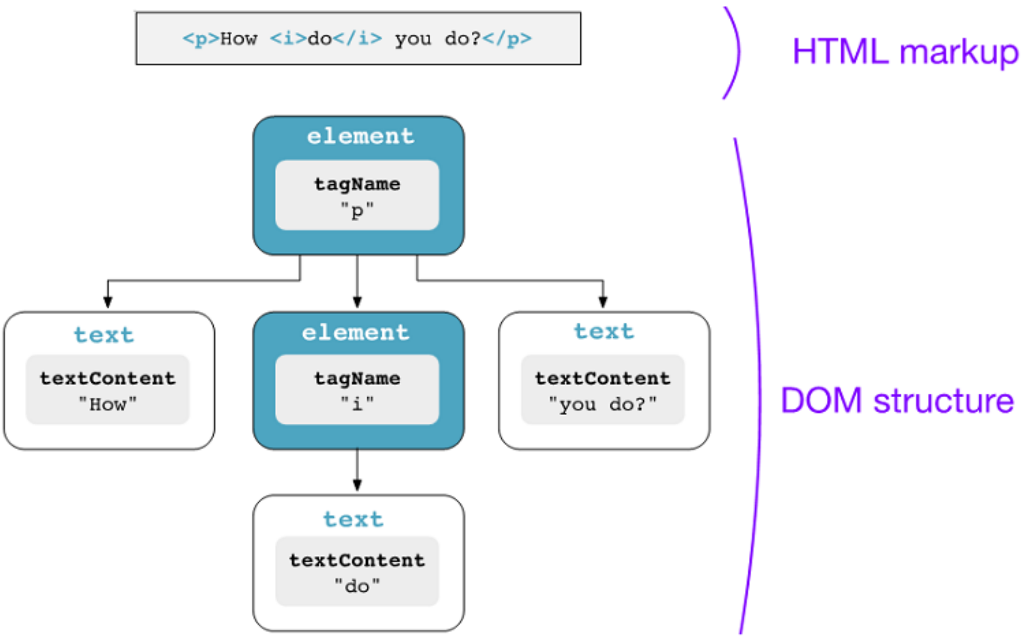

Figure 2.1 Intermingling element DOM nodes and text nodes

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 do not write markup. Instead, we call functions to create representations of individual 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.

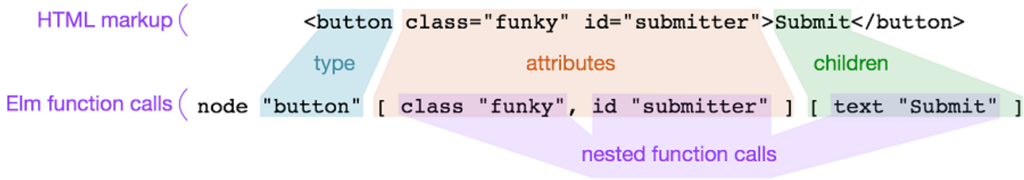

Figure 2.2 Representing a button using HTML markup (top) and Elm's node function (bottom)

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 plethora 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 helper functions like img as much as possible, and to fall back on node only in cases where no helper function is available. (For example, you may notice that there is no helper 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 bring up the *Try Elm* website at http://elm-lang.org/try and enter the following into the left-hand pane.

Listing 2.2 Building a view in *Try Elm*

```
import Html exposing (div, h1, img, text)        ❶
import Html.Attributes exposing (..)             ❶
```

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


main =  ❺
    view "no model yet"
```

❶ **We'll discuss "exposing" later**
❷ **h1 element with an empty attributes list**
❸ **Put commas at the start of the line**
❹ **img element with an empty children list**
❺ **We'll discuss "main" later**

Press **Compile**. The words "Photo Groove" should appear in the right-hand pane, followed by three images. (Depending on the size of your browser, the images may not be side-by-side.)

Figure 2.3 The Try Elm website

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

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

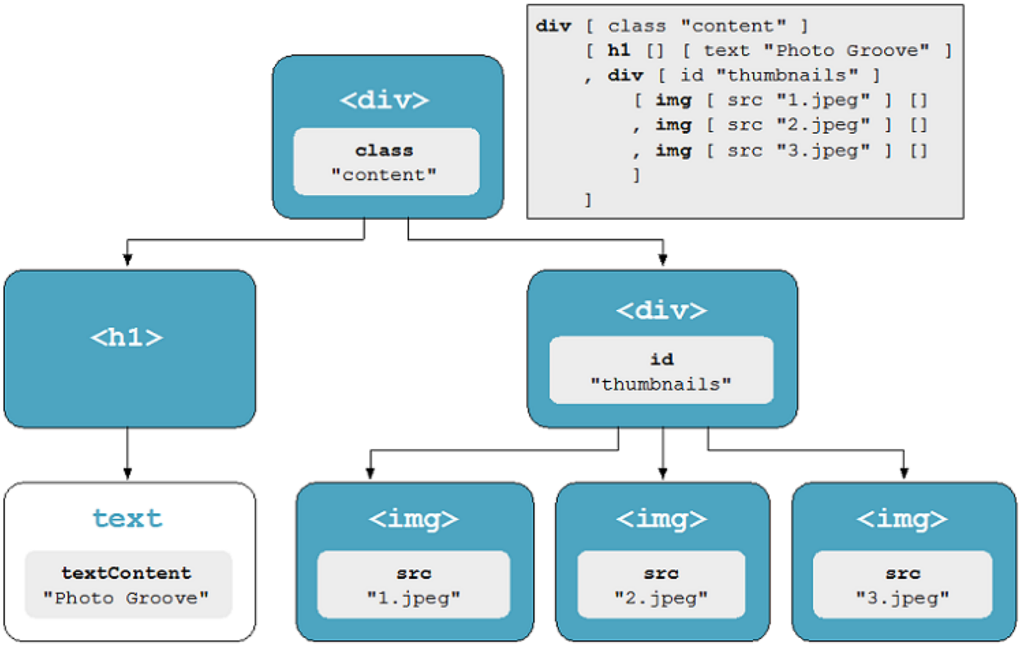

Figure 2.4 The DOM structure of your first Elm user interface

Notice how the helper 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 so:

```
br [] []
```

This pattern is consistent throughout the `Html` module, and it's worth following if you ever decide to make a custom element helper of your own using the `node` function.

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 a similar result by assigning them to constants:

```
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!

When to use qualified over unqualified imports

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 (..)` always?

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 (..)
> filter
```

You'll get an error saying that `filter` is ambiguous. After importing and exposing both `String.filter` and `List.filter`, 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.filter` or `List.filter` 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 http://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! We could style our elements inline using the `Html.Attribtues.style` attribute, but instead we'll organize things the traditional way: with our CSS declarations in a separate file.

The only way to get multiple files involved in the same page is to give a browser some HTML markup, so our first step in the process of styling our application will be to create an `index.html` file. Since *Try Elm* doesn't let you bring your own HTML file to the party, this means we'll need to start building our project locally.

1. Make a new directory called `PhotoGroove` and open it in a terminal.
2. Create a file called `PhotoGroove.elm` inside this directory.
3. Copy the *Try Elm* code from Listing 2.2 into `PhotoGroove.elm`.
4. Add the line `module PhotoGroove exposing (..)` to the top of `PhotoGroove.elm`.
5. Change `import Html exposing (div, h1, img, text)` to use `exposing (..)`.
6. When the dust settles, we should have a `PhotoGroove.elm` file that looks like Listing 2.3.

Listing 2.3 PhotoGroove.elm

```
module PhotoGroove exposing (..)    ❶

import Html exposing (..)   ❷
import Html.Attributes exposing (..)


view model =
```

```
    div [ class "content" ]
        [ h1 [] [ text "Photo Groove" ]
        , 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" ] []
            ]
        ]


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

❶ "This file describes a module called PhotoGroove"
❷ Remember: this uses "exposing (..)" now!
❸ This will be the application's entry point

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 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.

DECLARING THE PHOTOGROOVE MODULE

By writing `module PhotoGroove exposing (..)` at the top of our `PhotoGroove.elm` file, we defined a new module of our own. This means other files in our project can now 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 (..)` after `module PhotoGroove`, we are exposing all of our top-level values—namely `main` and `view`—for other modules to import.

If we wanted to expose `main` but leave `view` invisible to other modules, we could have written `exposing (main)` instead. In such a case, 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!

INSTALLING PACKAGE DEPENDENCIES

At this point we have a `PhotoGroove.elm` file, but it's missing a dependency. All this fancy rendering logic we've been writing comes from the `Html` module, but unlike the modules we've been using so far—like `String` and `List`—the `Html` module does not come preinstalled. To access it, we need to install the *package* that contains the `Html` module.

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

The `elm-package` utility 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-lang/html`.

Let's use `elm-package` to get some `Html` going. Run this command in the terminal:

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

You should see something like this:

```
Some new packages are needed. Here is the upgrade plan.
(it lists the packages here)
Do you approve of this plan? (Y/n)
```

Answer `y`, and you should shortly see the text `Packages configured successfully!` You should now have a file called `elm-package.json` in your current directory, which `elm-package` will have created for you automatically. We'll dig into `elm-package.json` in future chapters.

COMPILING TO JAVASCRIPT AND RUNNING THE APPLICATION

Now we're ready to compile our Elm code into JavaScript. Run this in the terminal:

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

This will compile our `PhotoGroove.elm` file into a JavaScript file we can give to a browser. (That generated JavaScript file will be called `elm.js`, because we passed `--output elm.js` to `elm-make`.) Now all we need is a HTML file that will load up our compiled `elm.js` file!

Make a file called `index.html` and put the following 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="elm-area"></div>   ❶

        <script src="elm.js"></script>   ❷
        <script>
            Elm.PhotoGroove.embed(document.getElementById("elm-area"));   ❸
        </script>
    </body>
</html>
```

❶ **Our Elm application will render into this div**
❷ **PhotoGroove.elm will get compiled into elm.js**
❸ **The Elm object comes from elm.js**

The markup we put in this file covers things like:

- The standard `<!doctype>`, `<html>`, and `<body>` tags
- Importing the `elm.js` file we generated with `elm-make`

- Whatever <head> inclusions we need—styles, metadata, <title>, and so on

The line Elm.PhotoGroove.embed(document.getElementById("elm-area")); starts our Elm code running in the <div id="elm-area"></div> element we set up earlier in index.html. Let's break down what happens when our Elm code starts running.

THE ELM RUNTIME AND 'MAIN'

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

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

When we called Elm.PhotoGroove.embed, we told the Elm Runtime to open the module called PhotoGroove and use its exposed main value as the entry point for the application. If we did not have a module called PhotoGroove, or if it did not expose a main value, we'd get 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.embed instead, but otherwise everything would still work. If the CubeDraft module did not expose a value called main, however, the application would not start. There's no renaming main!

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

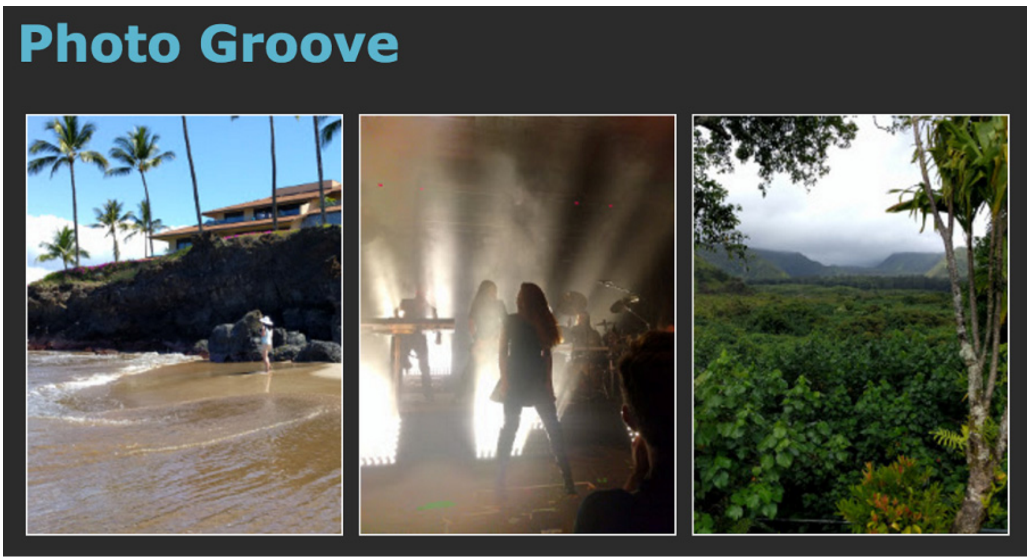

Figure 2.5 Rendering the application

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*—we just generated some `Html` and rendered it. That will soon change, as we're about to start handling user input! This brings us to a common question: 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.

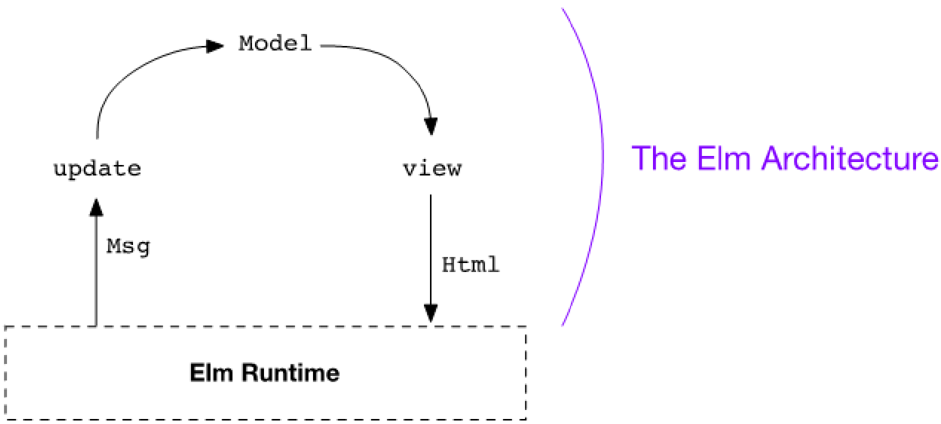

Figure 2.6 The Elm Runtime uses the Elm Architecture to manage data flow

Let's begin where data flow naturally begins: with application state as a whole.

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 and unit testing became increasingly important. 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 what we do in the Elm Architecture.

DECLARING A MODEL

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

> **DEFINITION** **A *model* stores the current state of an application. Any value necessary to render a user interface should be stored in its model.**

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" }   ❶
    , { url = "2.jpeg" }   ❶
    , { url = "3.jpeg" }   ❶
    ]


main =
    view initialModel
```

❶ **We'll add more fields beyond url later**

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. A typical Elm application does this through a *view function,* which describes how the DOM should look based on its arguments.

At the top 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 renders 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` function, which renders a single thumbnail as `Html`. We can the stage for that design by replacing our `view` implementation with the following:

Listing 2.6 Splitting out viewThumbnail

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


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


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

❶ **We'll prepend this to strings like "1.jpeg"**
❷ **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.

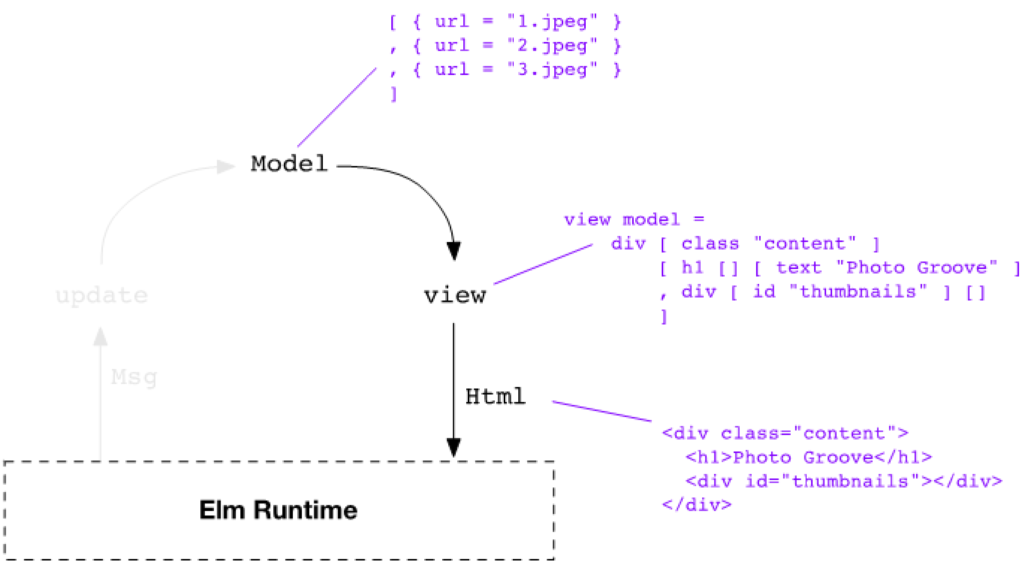

Figure 2.7 Model and view connecting with the Elm Runtime

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. You 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`.

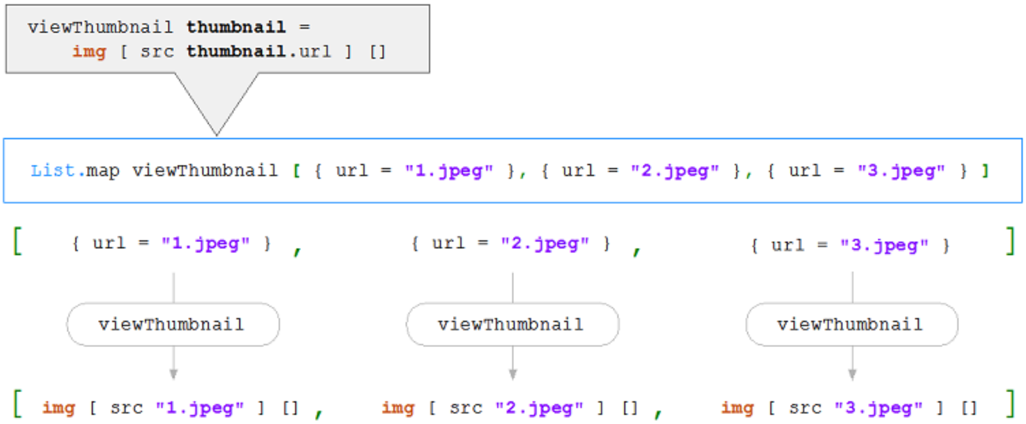

Figure 2.8 Using List.map to transform photo records into img nodes

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 go ahead and 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 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"   ❶
    }
```

❶ **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! Being a lowly helper function, `viewThumbnail` has no way to access the model—so it can't know the current value of `selectedUrl`. But without knowing which thumbnail is selected, how can it possibly 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    ❶
            [ class "large"
            , src (urlPrefix ++ "large/" ++ model.selectedUrl)
            ]
            []
        ]
```

❶ **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 providing one or more of its arguments, and getting back a new function which accepts the remaining arguments and finishes 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 aren't set up to support partial application by default. If we'd written `viewThumbnail` in JavaScript, and wanted it to support partial application, it would have 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 Tuples.)

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	`splitA separator str =`
	`    String.split separator str`
		`function splitA(sep) {`
		`  return function(str) {`
		`    return str.split(sep);`
		`  }`
		`}`

Tupled Function	`splitB ( separator, str ) =`
	`    String.split separator str`
		`function splitB(sep, str) {`
		`  return str.split(sep);`
		`}`

Total Application	`splitB ( '-', "867-5309" )`	`splitB('-', "867-5309")`
Total Application	`splitA   '-'   "867-5309"`	`splitA('-')("867-5309")`
Partial Application	`splitA   '-'`	`splitA('-')`

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)   ❶
        , img
            [ class "large"
            , src (urlPrefix ++ "large/" ++ model.selectedUrl)
            ]
            []
        ]
```

❶ **Partially apply viewThumbnail with model.selectedUrl**

Since all Elm functions are curried, it's common to give a helper 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!

> **TIP** Since operators are functions, you can partially apply them too! `List.map ((*) 2) [ 1, 2, 3 ]` evaluates to `[ 2, 4, 6 ]`.

Incidentally, currying is named after acclaimed logician Haskell Brooks Curry. The Haskell programming language is also named after his first name, and 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:

```
myThumbnail.addEventListener(function() { model.selectedUrl = myUrl; });
```

Elm wires up event handlers a bit differently. Similarly to how we wrote a `view` function that used Virtual DOM nodes to describe our desired page structure, we're now going to write an `update` function that uses *messages* to describe 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 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:

```
{ operation = "SELECT_PHOTO", data = "2.jpeg" }
```

This record is a message which conveys the following information:

> **"We should update our model to set** `2.jpeg` **as the** `selectedUrl`**."**

The `update` function receives this message and does the following:

1. Looks at the message it received.
2. Looks at our current model.
3. Uses these two values to determine a new model, then returns it.

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

```
update msg model =
    if msg.operation == "SELECT_PHOTO" 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 request that a `SELECT_PHOTO` message be sent to `update` whenever the user clicks a thumbnail, by adding an onClick attribute to `viewThumbnail`:

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

```
main =
    Html.App.beginnerProgram
        { model = initialModel
        , view = view
        , update = update
        }
```

The `Html.App.beginnerProgram` 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 `beginnerProgram` involved, `main` could only render static views. `beginnerProgram` lets us specify how we want to react to user input!

Figure 2.11 demonstrates how data flows through our revised application.

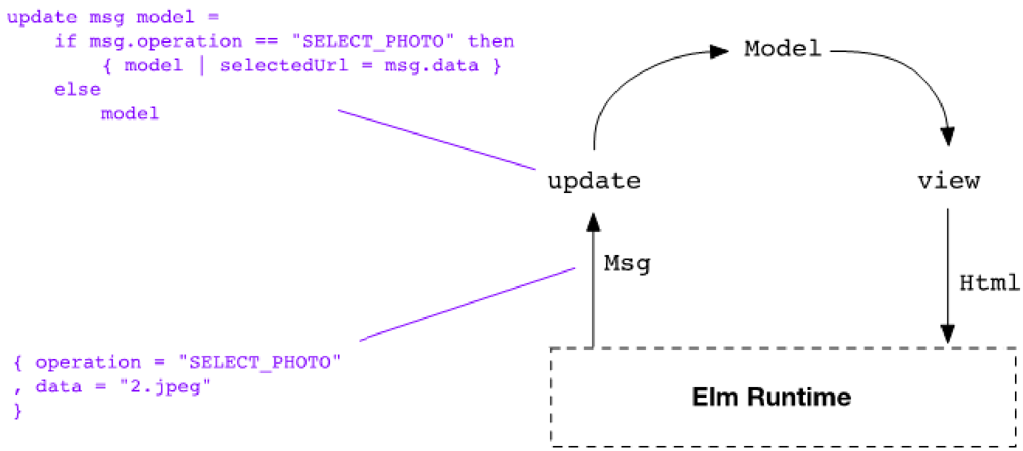

Figure 2.11 Data flowing from the start of the program through the Model-View-Update loop

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

Since `Html.App.beginnerProgram` lives in the `Html.App` module, and `onClick` lives in the `Html.Events` module, we'll need to `import` both of those modules:

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

And with that final touch...*it's alive!* You've now written an interactive Elm application!

The complete `PhotoGroove.elm` file should look like this:

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

```
module PhotoGroove exposing (..)

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


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


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


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


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


update msg model =
    if msg.operation == "SELECT_PHOTO" then
        { model | selectedUrl = msg.data }
    else
        model


main =
```

```
Html.App.beginnerProgram
    { model = initialModel
    , view = view
    , update = update
    }
```

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

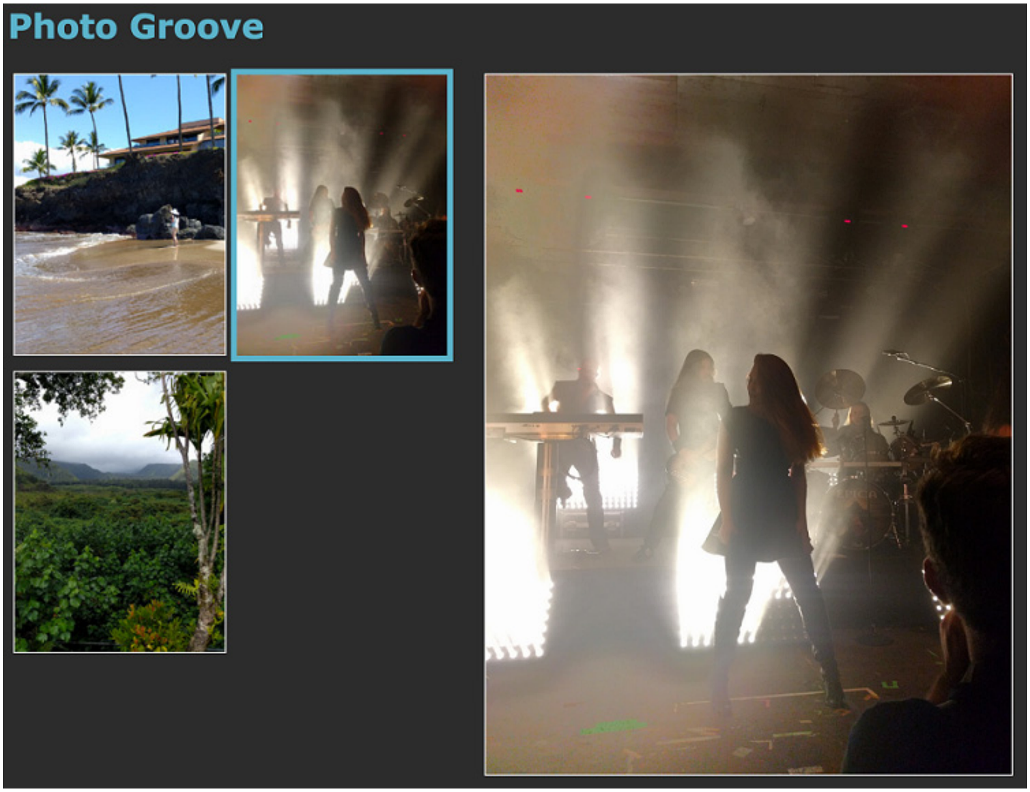

Figure 2.12 Our final Photo Groove application

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.

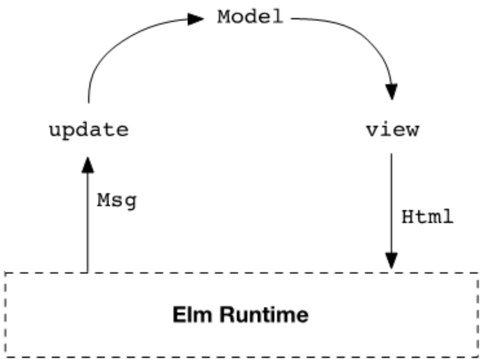

Figure 2.13 Our final Elm Architecture setup

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 `Html` 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.App.program` 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

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 more reliable and easier to maintain.

Onward!

A
Getting Set Up

This covers installing

- **Node.js (4.0 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 4.0 or higher and NPM 2.0 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 4.0 or higher and NPM 2.0 or higher installed, you should be able to run these commands in your terminal and see similar output:

```
$ node --version
v4.0.0

$ npm --version
2.0.0
```

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:

```
npm install -g elm
```

> **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, 2, and 3. Starting in Chapter 4, 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 0.17.1 (Elm Platform 0.17.1)
```

If you see a version higher than `0.17.1`, then as of this writing, you are living in the future! Hello from the past, where `0.17.1` is the latest release.

> **NOTE** Any version that starts with `0.17` should work fine with this book, but according to semantic versioning, a version number beginning with `0.18` or higher indicates breaking changes. In such a case, there's a good chance your examples will not compile! Try `npm install -g elm@0.17.1` 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!