CHAPTER 1

Introducing Functional JavaScript

This chapter sets up the book in a number of important ways. In it, I will introduce Underscore and explain how you can start using it. Additionally, I will define the terms and goals of the rest of the book.

The Case for JavaScript

The question of why you might choose JavaScript is easily answered in a word: reach. In other words, aside from perhaps Java, there is no more popular programming language right now than JavaScript. Its ubiquity in the browser and its near-ubiquity in a vast sea of current and emerging technologies make it a nice—and sometimes the only—choice for portability.

With the reemergence of client-service and single-page application architectures, the use of JavaScript in discrete applications (i.e., single-page apps) attached to numerous network services is exploding. For example, Google Apps are all written in JavaScript, and are prime examples of the single-page application paradigm.

If you've come to JavaScript with a ready interest in functional programming, then the good news is that it supports functional techniques "right out of the box" (e.g., the function is a core element in JavaScript). For example, if you have any experience with JavaScript, then you might have seen code like the following:

```
[1, 2, 3].forEach(alert);
// alert box with "1" pops up
// alert box with "2" pops up
// alert box with "3" pops up
```

The `Array#forEach` method, added in the fifth edition of the ECMA-262 language standard, takes some function (in this case, `alert`) and passes each array element to the function one after the other. That is, JavaScript provides various methods and functions that take other functions as arguments for some inner purpose. I'll talk more about this style of programming as the book progresses.

JavaScript is also built on a solid foundation of language primitives, which is amazing, but a double-edged sword (as I'll discuss soon). From functions to closures to prototypes to a fairly nice dynamic core, JavaScript provides a well-stocked set of tools.[1] In addition, JavaScript provides a very open and flexible execution model. As a small example, all JavaScript functions have an `apply` method that allows you to call the function with an array as if the array elements were the arguments to the function itself. Using `apply`, I can create a neat little function named `splat` that just takes a function and returns another function that takes an array and calls the original with `apply`, so that its elements serve as its arguments:

```
function splat(fun) {
  return function(array) {
    return fun.apply(null, array);
  };
}

var addArrayElements = splat(function(x, y) { return x + y });

addArrayElements([1, 2]);
//=> 3
```

This is your first taste of functional programming—a function that returns another function—but I'll get to the meat of that later. The point is that `apply` is only one of many ways that JavaScript is a hugely flexible programming language.

Another way that JavaScript proves its flexibility is that any function may be called with any number of arguments of any type, at any time. We can create a function `unsplat` that works opposite from `splat`, taking a function and returning another function that takes any number of arguments and calls the original with an array of the values given:

```
function unsplat(fun) {
  return function() {
    return fun.call(null, _.toArray(arguments));
  };
}

var joinElements = unsplat(function(array) { return array.join(' ') });

joinElements(1, 2);
//=> "1 2"

joinElements('-', '$', '/', '!', ':');
//=> "- $ / ! :"
```

1. And, as with all tools, you can get cut and/or smash your thumb if you're not careful.

Every JavaScript function can access a local value named `arguments` that is an array-like structure holding the values that the function was called with. Having access to `arguments` is surprisingly powerful, and is used to amazing effect in JavaScript in the wild. Additionally, the `call` method is similar to `apply` except that the former takes the arguments one by one rather than as an array, as expected by `apply`. The trifecta of `apply`, `call`, and `arguments` is only a small sample of the extreme flexibility provided by JavaScript.

With the emergent growth of JavaScript for creating applications of all sizes, you might expect stagnation in the language itself or its runtime support. However, even a casual investigation of the `ECMAScript.next` initiative shows that it's clear that JavaScript is an evolving (albeit slowly) language.[2] Likewise, JavaScript engines like V8 are constantly evolving and improving JavaScript speed and efficiency using both time-tested and novel techniques.

Some Limitations of JavaScript

The case against JavaScript—in light of its evolution, ubiquity, and reach—is quite thin. You can say much about the language quirks and robustness failings, but the fact is that JavaScript is here to stay, now and indefinitely. Regardless, it's worth acknowledging that JavaScript is a flawed language.[3] In fact, the most popular book on JavaScript, Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly), spends more pages discussing the terrible parts than the good. The language has true oddities, and by and large is not particularly succinct in expression. However, changing the problems with JavaScript would likely "break the Web," a circumstance that's unacceptable to most. It's because of these problems that the number of languages targeting JavaScript as a compilation platform is growing; indeed, this is a very fertile niche.[4]

As a language supporting—and at times preferring—imperative programming techniques and a reliance on global scoping, JavaScript is unsafe by default. That is, building programs with a key focus on mutability is potentially confusing as programs grow. Likewise, the very language itself provides the building blocks of many high-level features found by default in other languages. For example, JavaScript itself, prior to trunk versions of ECMAScript 6, provides no module system, but facilitates their creation using raw objects. That JavaScript provides a loose collection of basic parts ensures a bevy of custom module implementations, each incompatible with the next.

2. A draft specification for `ES.next` is found at *http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts*.

3. The debate continues over just how deeply.

4. Some languages that target JavaScript include, but are not limited to, the following: ClojureScript, CoffeeScript, Roy, Elm, TypeScript, Dart, Flapjax, Java, and JavaScript itself!

Language oddities, unsafe features, and a sea of competing libraries: three legitimate reasons to think hard about the adoption of JavaScript. But there is a light at the end of the tunnel that's not just the light of an oncoming train. The light is that through discipline and an observance to certain conventions, JavaScript code can be not only safe, but also simple to understand and test, in addition to being proportionally scalable to the size of the code base. This book will lead you on the path to one such approach: functional programming.

Getting Started with Functional Programming

You may have heard of functional programming on your favorite news aggregation site, or maybe you've worked in a language supporting functional techniques. If you've written JavaScript (and in this book I assume that you have) then you indeed *have* used a language supporting functional programming. However, that being the case, you might not have used JavaScript in a functional way. This book outlines a functional style of programming that aims to simplify your own libraries and applications, and helps tame the wild beast of JavaScript complexity.

As a bare-bones introduction, functional programming can be described in a single sentence:

> Functional programming is the use of functions that transform values into units of abstraction, subsequently used to build software systems.

This is a simplification bordering on libel, but it's functional (ha!) for this early stage in the book. The library that I use as my medium of functional expression in JavaScript is Underscore, and for the most part, it adheres to this basic definition. However, this definition fails to explain the "why" of functional programming.

Why Functional Programming Matters

> The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.
>
> —John Carmack

If you're familiar with object-oriented programming, then you may agree that its primary goal is to break a problem into parts, as shown in Figure 1-1 (Gamma 1995).

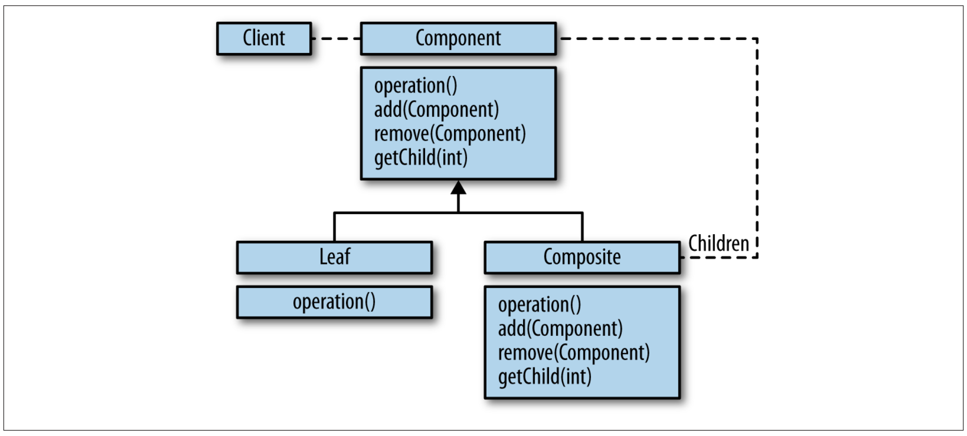

Figure 1-1. A problem broken into object-oriented parts

Likewise, these parts/objects can be aggregated and composed to form larger parts, as shown in Figure 1-2.

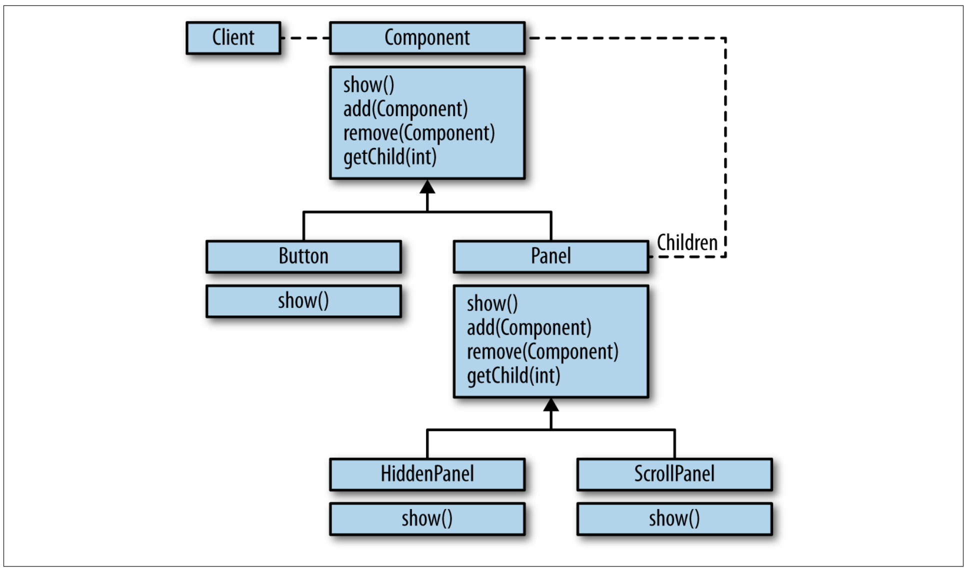

Figure 1-2. Objects are "composed" together to form bigger objects

Based on these parts and their aggregates, a system is then described in terms of the interactions and values of the parts, as shown in Figure 1-3.

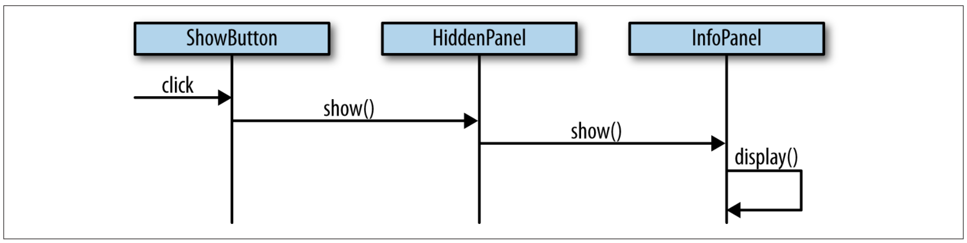

Figure 1-3. An object-oriented system and its interactions as a sequence diagram

This is a gross simplification of how object-oriented systems are formed, but I think that as a high-level description it works just fine.

By comparison, a strict functional programming approach to solving problems also breaks a problem into parts (namely, functions), as shown in Figure 1-4.

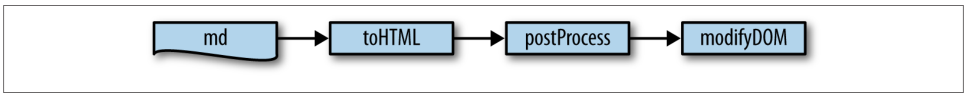

Figure 1-4. A problem broken into functional parts

Whereas the object-oriented approach tends to break problems into groupings of "nouns," or objects, a functional approach breaks the same problem into groupings of "verbs," or functions.[5] As with object-oriented programming, larger functions are formed by "gluing" or "composing" other functions together to build high-level behaviors, as shown in Figure 1-5.

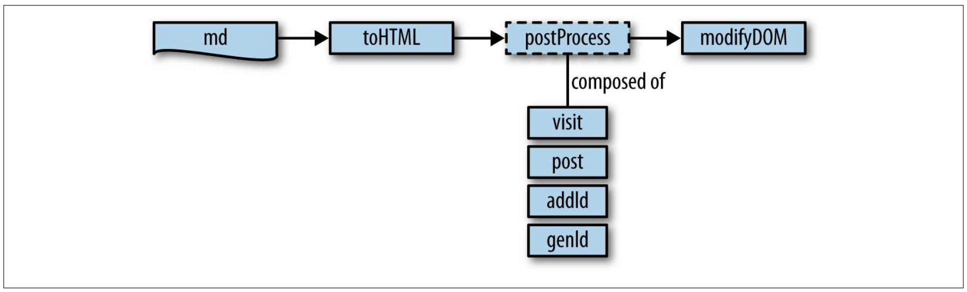

Figure 1-5. Functions are also composed together to form more behaviors

5. This is a simplistic way to view the composition of object-oriented versus functional systems, but bear with me as I develop a way to mix the two throughout the course of this book.

Finally, one way that the functional parts are formed into a system (as shown in Figure 1-6) is by taking a value and gradually "transforming" it—via one primitive or composed function—into another.

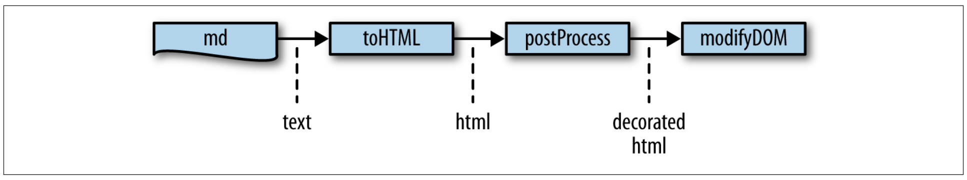

Figure 1-6. A functional system interacts via data transformation

In a system observing a strict object-oriented style, the interactions between objects cause internal change to each object, leading to an overall system state that is the amalgamation of many smaller, potentially subtle state changes. These interrelated state changes form a conceptual "web of change" that, at times, can be confusing to keep in your head. This confusion becomes a problem when the act of adding new objects and system features requires a working knowledge of the subtleties of potentially far-reaching state changes.

A functional system, on the other hand, strives to minimize observable state modification. Therefore, adding new features to a system built using functional principles is a matter of understanding how new functions can operate within the context of localized, nondestructive (i.e., original data is never changed) data transformations. However, I hesitate to create a false dichotomy and say that functional and object-oriented styles should stand in opposition. That JavaScript supports both models means that systems can and should be composed of both models. Finding the balance between functional and object-oriented styles is a tricky task that will be tackled much later in the book, when discussing mixins in Chapter 9. However, since this is a book about functional programming in JavaScript, the bulk of the discussion is focused on functional styles rather than object-oriented ones.

Having said that, a nice image of a system built along functional principles is an assembly-line device that takes raw materials in one end, and gradually builds a product that comes out the other end (Figure 1-7).

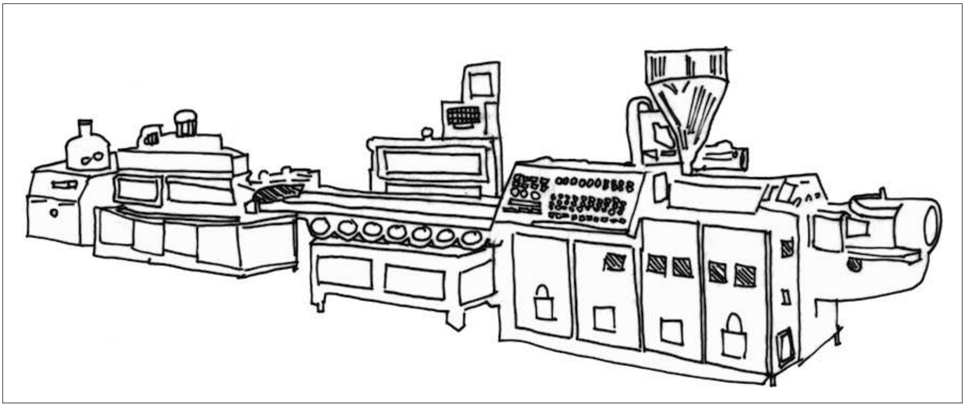

Figure 1-7. A functional program is a machine for transforming data

The assembly line analogy is, of course, not entirely perfect, because every machine I know consumes its raw materials to produce a product. By contrast, functional programming is what happens when you take a system built in an imperative way and shrink explicit state changes to the smallest possible footprint to make it more modular (Hughes 1984). Practical functional programming is not about eliminating state change, but instead about reducing the occurrences of mutation to the smallest area possible for any given system.

Functions as Units of Abstraction

One method of abstraction is that functions hide implementation details from view. In fact, functions are a beautiful unit of work allowing you to adhere to the long-practiced maxim in the UNIX community, set forth by Butler Lampson:

> Make it run, make it right, make it fast.

Likewise, functions-as-abstraction allow you to fulfill Kent Beck's similarly phrased mantra of test-driven development (TDD):

> Make it run, then make it right, then make it fast.

For example, in the case of reporting errors and warnings, you could write something like the following:

```
function parseAge(age) {
  if (!_.isString(age)) throw new Error("Expecting a string");
  var a;

  console.log("Attempting to parse an age");

  a = parseInt(age, 10);
  if (_.isNaN(a)) {
    console.log(["Could not parse age:", age].join(' '));
    a = 0;
  }

  return a;
}
```

This function, although not comprehensive for parsing age strings, is nicely illustrative. Use of `parseAge` is as follows:

```
parseAge("42");
// (console) Attempting to parse an age
//=> 42

parseAge(42);
// Error: Expecting a string

parseAge("frob");
// (console) Attempting to parse an age
// (console) Could not parse age: frob
//=> 0
```

The `parseAge` function works as written, but if you want to modify the way that errors, information, and warnings are presented, then changes need to be made to the appropriate lines therein, and anywhere else similar patterns are used. A better approach is to "abstract" the notion of errors, information, and warnings into functions:

```
function fail(thing) {
  throw new Error(thing);
}

function warn(thing) {
  console.log(["WARNING:", thing].join(' '));
}

function note(thing) {
  console.log(["NOTE:", thing].join(' '));
}
```

Using these functions, the `parseAge` function can be rewritten as follows:

```
function parseAge(age) {
  if (!_.isString(age)) fail("Expecting a string");
  var a;

  note("Attempting to parse an age");
  a = parseInt(age, 10);

  if (_.isNaN(a)) {
    warn(["Could not parse age:", age].join(' '));
    a = 0;
  }

  return a;
}
```

Here's the new behavior:

```
parseAge("frob");
// (console) NOTE: Attempting to parse an age
// (console) WARNING: Could not parse age: frob
//=> 0
```

It's not very different from the old behavior, except that now the idea of reporting errors, information, and warnings has been abstracted away. The reporting of errors, information, and warnings can thus be modified entirely:

```
function note() {}
function warn(str) {
  alert("That doesn't look like a valid age");
}

parseAge("frob");
// (alert box) That doesn't look like a valid age
//=> 0
```

Therefore, because the behavior is contained within a single function, the function can be replaced by new functions providing similar behavior or outright different behaviors altogether (Abelson and Sussman 1996).

Encapsulation and Hiding

Over the years, we've been taught that a cornerstone of object-oriented programming is *encapsulation*. The term encapsulation in reference to object-oriented programming refers to a way of packaging certain pieces of data with the very operations that manipulate them, as seen in Figure 1-8.

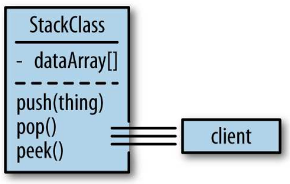

Figure 1-8. Most object-oriented languages use object boundaries to package data elements with the operations that work on them; a Stack class would therefore package an array of elements with the push, pop, and peek operations used to manipulate it

JavaScript provides an object system that does indeed allow you to encapsulate data with its manipulators. However, sometimes encapsulation is used to restrict the visibility of certain elements, and this act is known as *data hiding*. JavaScript's object system does not provide a way to hide data directly, so data is hidden using something called closures, as shown in Figure 1-9.

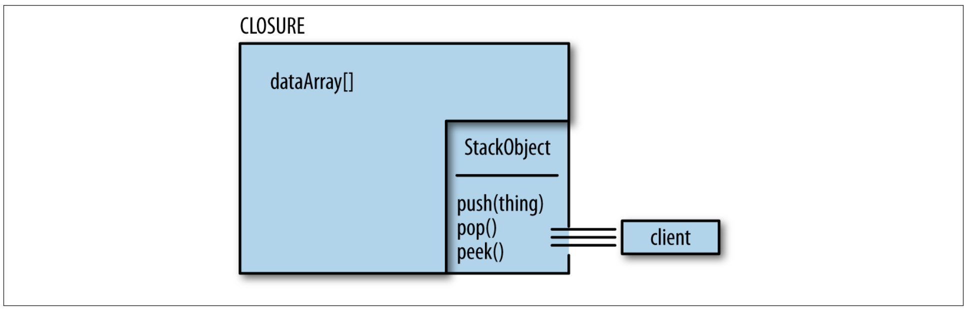

Figure 1-9. Using a closure to encapsulate data is a functional way to hide details from a client's view

Closures are not covered in any depth until Chapter 3, but for now you should keep in mind that closures are kinds of functions. By using functional techniques involving closures, you can achieve data hiding that is as effective as the same capability offered by most object-oriented languages, though I hesitate to say whether functional encapsulation or object-oriented encapsulation is better. Instead, while they are different in practice, they both provide similar ways of building certain kinds of abstraction. In fact, this book is not at all about encouraging you to throw away everything that you might have ever learned in favor of functional programming; instead, it's meant to explain functional programming on its own terms so that you can decide if it's right for your needs.

Functions as Units of Behavior

Hiding data and behavior (which has the side effect of providing a more agile change experience) is just one way that functions can be units of abstraction. Another is to provide an easy way to store and pass around discrete units of basic behavior. Take, for example, JavaScript's syntax to denote looking up a value in an array by index:

```
var letters = ['a', 'b', 'c'];

letters[1];
//=> 'b'
```

While array indexing is a core behavior of JavaScript, there is no way to grab hold of the behavior and use it as needed without placing it into a function. Therefore, a simple example of a function that abstracts array indexing behavior could be called `nth`. The naive implementation of `nth` is as follows:

```
function naiveNth(a, index) {
  return a[index];
}
```

As you might suspect, `nth` operates along the happy path perfectly fine:

```
naiveNth(letters, 1);
//=> "b"
```

However, the function will fail if given something unexpected:

```
naiveNth({}, 1);
//=> undefined
```

Therefore, if I were to think about the abstraction surrounding a function `nth`, I might devise the following statement: *nth returns the element located at a valid index within a data type allowing indexed access*. A key part of this statement is the idea of an indexed data type. To determine if something is an indexed data type, I can create a function `isIndexed`, implemented as follows:

```
function isIndexed(data) {
  return _.isArray(data) || _.isString(data);
}
```

The function `isIndexed` is also a function providing an abstraction over checking if a piece of data is a string or an array. Building abstraction on abstraction leads to the following complete implementation of `nth`:

```
function nth(a, index) {
  if (!_.isNumber(index)) fail("Expected a number as the index");
  if (!isIndexed(a)) fail("Not supported on non-indexed type");
  if ((index < 0) || (index > a.length - 1))
    fail("Index value is out of bounds");

  return a[index];
}
```

The completed implementation of `nth` operates as follows:

```
nth(letters, 1);
//=> 'b'

nth("abc", 0);
//=> "a"

nth({}, 2);
// Error: Not supported on non-indexed type

nth(letters, 4000);
// Error: Index value is out of bounds
```

```
nth(letters, 'aaaaa');
// Error: Expected a number as the index
```

In the same way that I built the `nth` abstraction out of an `indexed` abstraction, I can likewise build a `second` abstraction:

```
function second(a) {
  return nth(a, 1);
}
```

The `second` function allows me to appropriate the correct behavior of `nth` for a different but related use case:

```
second(['a','b']);
//=> "b"

second("fogus");
//=> "o"

second({});
// Error: Not supported on non-indexed type
```

Another unit of basic behavior in JavaScript is the idea of a *comparator*. A comparator is a function that takes two values and returns `<1` if the first is less than the second, `>1` if it is greater, and `0` if they are equal. In fact, JavaScript itself can appear to use the very nature of numbers themselves to provide a default `sort` method:

```
[2, 3, -6, 0, -108, 42].sort();
//=> [-108, -6, 0, 2, 3, 42]
```

But a problem arises when you have a different mix of numbers:

```
[0, -1, -2].sort();
//=> [-1, -2, 0]

[2, 3, -1, -6, 0, -108, 42, 10].sort();
//=> [-1, -108, -6, 0, 10, 2, 3, 42]
```

The problem is that when given no arguments, the `Array#sort` method does a string comparison. However, every JavaScript programmer knows that `Array#sort` expects a comparator, and instead writes:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(function(x,y) {
  if (x < y) return -1;
  if (y < x) return  1;
  return 0;
});

//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

That seems better, but there is a way to make it more generic. After all, you might need to sort like this again in another part of the code, so perhaps it's better to pull out the anonymous function and give it a name:

```
function compareLessThanOrEqual(x, y) {
  if (x < y) return -1;
  if (y < x) return  1;
  return 0;
}

[2, 3, -1, -6, 0, -108, 42, 10].sort(compareLessThanOrEqual);
//=> [-108, -6, -1, 0, 2, 3, 10, 42]
```

But the problem with the `compareLessThanOrEqual` function is that it is coupled to the idea of "comparatorness" and cannot easily stand on its own as a generic comparison operation:

```
if (compareLessThanOrEqual(1,1))
  console.log("less or equal");

// nothing prints
```

To achieve the desired effect, I would need to *know* about `compareLessThanOrEqual`'s comparator nature:

```
if (_.contains([0, -1], compareLessThanOrEqual(1,1)))
  console.log("less or equal");

// less or equal
```

But this is less than satisfying, especially when there is a possibility for some developer to come along in the future and change the return value of `compareLessThanOrEqual` to `-42` for negative comparisons. A better way to write `compareLessThanOrEqual` might be as follows:

```
function lessOrEqual(x, y) {
  return x <= y;
}
```

Functions that always return a Boolean value (i.e., `true` or `false` only), are called *predicates*. So, instead of an elaborate comparator construction, `lessOrEqual` is simply a "skin" over the built-in <= operator:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(lessOrEqual);
//=> [100, 10, 1, 0, -1, -1, -2]
```

At this point, you might be inclined to change careers. However, upon further reflection, the result makes sense. If `sort` expects a comparator, and `lessThan` only returns `true` or `false`, then you need to somehow get from the world of the latter to that of the former without duplicating a bunch of `if/then/else` boilerplate. The solution lies in creating a function, `comparator`, that takes a predicate and converts its result to the `-1/0/1` result expected of `comparator` functions:

```
function comparator(pred) {
  return function(x, y) {
    if (truthy(pred(x, y)))
```

```
          return -1;
        else if (truthy(pred(y, x)))
          return 1;
        else
          return 0;
      };
    };
```

Now, the `comparator` function can be used to return a new function that "maps" the results of the predicate `lessOrEqual` (i.e., `true` or `false`) onto the results expected of comparators (i.e., `-1`, `0`, or `1`), as shown in Figure 1-10.

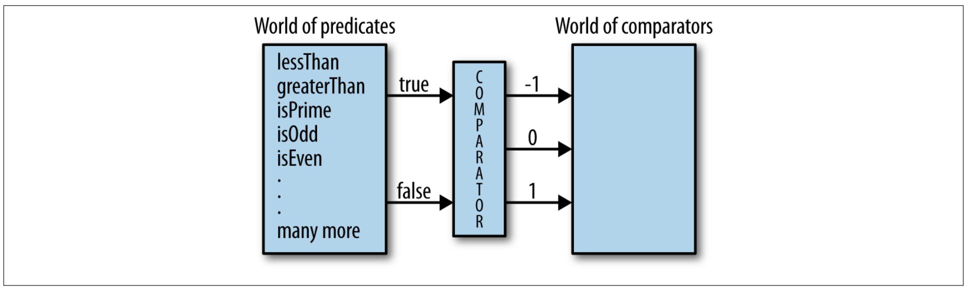

Figure 1-10. Bridging the gap between two "worlds" using the comparator function

In functional programming, you'll almost always see functions interacting in a way that allows one type of data to be brought into the world of another type of data. Observe `comparator` in action:

```
    [100, 1, 0, 10, -1, -2, -1].sort(comparator(lessOrEqual));
    //=> [-2, -1, -1, 0, 1, 10, 100]
```

The function `comparator` will work to map any function that returns "truthy" or "falsey" values onto the notion of "comparatorness." This topic is covered in much greater depth in Chapter 4, but it's worth noting now that `comparator` is a *higher-order function* (because it takes a function and returns a new function). Keep in mind that not every predicate makes sense for use with the `comparator` function, however. For example, what does it mean to use the `_.isEqual` function as the basis for a `comparator`? Try it out and see what happens.

Throughout this book, I will talk about the ways that functional techniques provide and facilitate the creation of abstractions, and as I'll discuss next, there is a beautiful synergy between functions-as-abstraction and data.

Data as Abstraction

JavaScript's object prototype model is a rich and foundational data scheme. On its own, the prototype model provides a level of flexibility not found in many other mainstream programming languages. However, many JavaScript programmers, as is their wont, immediately attempt to build a class-based object system using the prototype or closure features (or both).[6] Although a class system has its strong points, very often the data needs of a JavaScript application are much simpler than is served by classes.[7]

Instead, using JavaScript bare data primitives, objects, and arrays, much of the data modeling tasks that are currently served by classes are subsumed. Historically, functional programming has centered around building functions that work to achieve higher-level behaviors and work on very simple data constructs.[8] In the case of this book (and Underscore itself), the focus is indeed on processing arrays and objects. The flexibility in those two simple data types is astounding, and it's unfortunate that they are often overlooked in favor of yet another class-based system.

Imagine that you're tasked with writing a JavaScript application that deals with comma-separated value (CSV) files, which are a standard way to represent data tables. For example, suppose you have a CSV file that looks as follows:

```
name,   age, hair
Merble, 35,  red
Bob,    64,  blonde
```

It should be clear that this data represents a table with three columns (name, age, and hair) and three rows (the first being the header row, and the rest being the data rows). A small function to parse this very constrained CSV representation stored in a string is implemented as follows:

```
function lameCSV(str) {
  return _.reduce(str.split("\n"), function(table, row) {
    table.push(_.map(row.split(","), function(c) { return c.trim()}));
    return table;
  }, []);
};
```

You'll notice that the function `lameCSV` processes the rows one by one, splitting at `\n` and then stripping whitespace for each cell therein.[9] The whole data table is an array of sub-arrays, each containing strings. From the conceptual view shown in Table 1-1, nested arrays can be viewed as a table.

6. The `ECMAScript.next` initiative is discussing the possibility of language support for classes. However, for various reasons outside the scope of this book, the feature is highly controversial. As a result, it's unclear when and if classes will make it into JavaScript core.

7. One strong argument for a class-based object system is the historical use in implementing user interfaces.

8. Very often you will see a focus on list data structures in functional literature. In the case of JavaScript, the array is a nice substitute.

9. The function `lameCSV` is meant for illustrative purposes and is in no way meant as a fully featured CSV parser.

Table 1-1. Simply nested arrays are one way to abstract a data table

name	age	hair
Merble	35	red
Bob	64	blonde

Using `lameCSV` to parse the data stored in a string works as follows:

```
var peopleTable = lameCSV("name,age,hair\nMerble,35,red\nBob,64,blonde");

peopleTable;
//=> [["name",   "age",   "hair"],
//    ["Merble", "35",    "red"],
//    ["Bob",    "64",    "blonde"]]
```

Using selective spacing highlights the table nature of the returned array. In functional programming, functions like `lameCSV` and the previously defined `comparator` are key in translating one data type into another. Figure 1-11 illustrates how data transformations in general can be viewed as getting from one "world" into another.

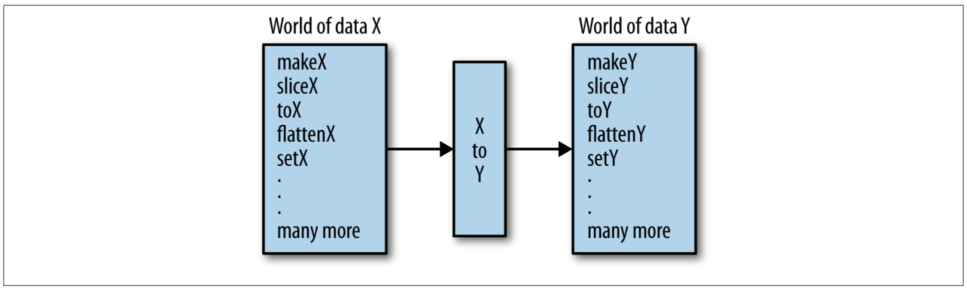

Figure 1-11. Functions can bridge the gap between two "worlds"

There are better ways to represent a table of such data, but this nested array serves us well for now. Indeed, there is little motivation to build a complex class hierarchy representing either the table itself, the rows, people, or whatever. Instead, keeping the data representation minimal allows me to use existing array fields and array processing functions and methods out of the box:

```
_.rest(peopleTable).sort();

//=> [["Bob",    "64",   "blonde"],
//    ["Merble", "35",   "red"]]
```

Likewise, since I know the form of the original data, I can create appropriately named selector functions to access the data in a more descriptive way:

```
function selectNames(table) {
  return _.rest(_.map(table, _.first));
}
```

```
function selectAges(table) {
  return _.rest(_.map(table, second));
}

function selectHairColor(table) {
  return _.rest(_.map(table, function(row) {
    return nth(row, 2);
  }));
}

var mergeResults = _.zip;
```

The `select` functions defined here use existing array processing functions to provide fluent access to simple data types:

```
selectNames(peopleTable);
//=> ["Merble", "Bob"]

selectAges(peopleTable);
//=> ["35", "64"]

selectHairColor(peopleTable);
//=> ["red", "blonde"]

mergeResults(selectNames(peopleTable), selectAges(peopleTable));
//=> [["Merble", "35"], ["Bob", "64"]]
```

The simplicity of implementation and use is a compelling argument for using JavaScript's core data structures for data modeling purposes. That's not to say that there is no place for an object-oriented or class-based approach. In my experience, I've found that a functional approach centered around generic collection processing functions is ideal for handling data about people and an object-oriented approach works best for simulating people.[10]

If you are so inclined, the data table could be changed to a custom class-based model, and as long as you use the selector abstractions, then the user would never know, nor care. However, throughout this book, I strive to keep the data needs as simple as possible and build abstract functions that operate on them. Constraining myself to functions operating on simple data, interestingly enough, increases my flexibility. You might be surprised how far these fundamental types will take you.

10. That the object-oriented paradigm sprung from the simulation community in the form of the Simula programming language is no coincidence. Having written my share of simulation systems, I feel very strongly that object orientation or actor-based modeling are compelling fits for simula.

A Taste of Functional JavaScript

This is not a book about navigating around the quirks of JavaScript. There are many other books that will help you along that path. However, before I start any JavaScript project these days, I define two useful functions that I often find a need for: `existy` and `truthy`.

The function `existy` is meant to define the existence of something. JavaScript has two values—`null` and `undefined`—that signify nonexistence. Thus, `existy` checks that its argument is neither of these things, and is implemented as follows:

```
function existy(x) { return x != null };
```

Using the loose inequality operator (`!=`), it is possible to distinguish between `null`, `undefined`, and everything else. It's used as follows:

```
existy(null);
//=> false

existy(undefined);
//=> false

existy({}.notHere);
//=> false

existy((function(){})());
//=> false

existy(0);
//=> true

existy(false);
//=> true
```

The use of `existy` simplifies what it means for something to exist in JavaScript. Minimally, it collocates the existence check in an easy-to-use function. The second function mentioned, `truthy`, is defined as follows:[11]

```
function truthy(x) { return (x !== false) && existy(x) };
```

The `truthy` function is used to determine if something should be considered a synonym for `true`, and is used as shown here:[12]

11. You might come across the idea of JavaScript's truthiness referring to the true-ness and false-ness of the language's native types. Although it's important to know the details of what constitutes truth for JavaScript, I like to simplify matters by reducing the number of rules that I need to consider for my own applications.

12. That the number zero is considered "truthy" is by design. That it is often used as a synonym for `false` is a relic of a C heritage. If you wish to retain this behavior, then simply do not use `truthy` where you might expect `0`.

```
truthy(false);
//=> false

truthy(undefined);
//=> false

truthy(0);
//=> true

truthy('');
//=> true
```

In JavaScript, it's sometimes useful to perform some action only if a condition is true and return something like `undefined` or `null` otherwise. The general pattern is as follows:

```
{
  if(condition)
    return _.isFunction(doSomething) ? doSomething() : doSomething;
  else
    return undefined;
}
```

Using `truthy`, I can encapsulate this logic in the following way:

```
function doWhen(cond, action) {
  if(truthy(cond))
    return action();
  else
    return undefined;
}
```

Now whenever that pattern rears its ugly head, you can do the following instead:[13]

```
function executeIfHasField(target, name) {
  return doWhen(existy(target[name]), function() {
    var result = _.result(target, name);
    console.log(['The result is', result].join(' '));
    return result;
  });
}
```

The execution of `executeIfHasField` for success and error cases is as follows:

```
executeIfHasField([1,2,3], 'reverse');
// (console) The result is 3, 2, 1
//=> [3, 2, 1]

executeIfHasField({foo: 42}, 'foo');
// (console) The result is 42
```

13. I use `existy(target[name])` rather than Underscore's `has(target, name)` because the latter will only check self-keys.

```
//=> 42

executeIfHasField([1,2,3], 'notHere');
//=> undefined
```

Big deal, right? So I've defined two functions—this is hardly functional programming. The functional part comes from their use. You may be familiar with the `Array#map` method available in many JavaScript implementations. It's meant to take a function and call it for every element in an array, returning a new array with the new values. It's used as follows:

```
[null, undefined, 1, 2, false].map(existy);
//=> [false, false, true, true, true]

[null, undefined, 1, 2, false].map(truthy);
//=> [false, false, true, true, false]
```

This, ladies and gentlemen, is functional programming:

- The definition of an abstraction for "existence" in the guise of a function
- The definition of an abstraction for "truthiness" built from existing functions
- The use of said functions by other functions via parameter passing to achieve some behavior

This book is all about code like this, but to an exquisite level of detail.

On Speed

I know what you're thinking. This functional programming stuff has to be slow as a dog, right?

There's no way to deny that the use of the array index form `array[0]` will execute faster than either of `nth(array, 0)` or `_.first(array)`. Likewise, an imperative loop of the following form will be very fast:[14]

```
for (var i=0, len=array.length; i < len; i++) {
  doSomething(array[i]);
}
```

An analogous use of Underscore's `_.each` function will, all factors being equal, be slower:

```
_.each(array, function(elem) {
  doSomething(array[i]);
});
```

However, it's very likely that all factors will not be equal. Certainly, if you had a function that needed speed, then a reasonable manual transformation would be to convert the

14. A wonderful site that I use for JavaScript browser benchmarking is located at *http://www.jsperf.com*.

internal use of `_.each` into an analogous use of `for` or `while`. Happily, the days of the ponderously slow JavaScript are coming to an end, and in some cases are a thing of the past. For example, the release of Google's V8 engine ushered in an age of runtime optimizations that have worked to motivate performance gains across all JavaScript engine vendors (Bak 2012).[15] Even if other vendors were not following Google's lead, the prevalence of the V8 engine is growing, and in fact drives the very popular Chrome browser and Node.js itself. However, other vendors are following the V8 lead and introducing runtime speed enhancements such as native-code execution, just-in-time compilation, faster garbage collection, call-site caching, and in-lining into their own JavaScript engines.[16]

However, the need to support aging browsers like Internet Explorer 6 is a very real requirement for some JavaScript programmers. There are two factors to consider when confronted with legacy platforms: (1) the use of IE6 and its ilk is dying out, and (2) there are other ways to gain speed before the code ever hits the browser.[17] For example, the subject of in-lining is particularly interesting, because many in-lining optimizations can be performed *statically*, or before code is ever run. Code in-lining is the act of taking a piece of code contained in, say, a function, and "pasting" it in place of a call to that very function. Let's take a look at an example to make things clearer. Somewhere in the depths of Underscore's `_.each` implementation is a loop very similar to the `for` loop shown earlier (edited for clarity):

```
_.each = function(obj, iterator, context) {
  // bounds checking
  // check for native method
  // check for length property
    for (var i = 0, l = obj.length; i < l; i++) {
      // call the given function
    }
}
```

Imagine that you have a bit of code that looks as follows:

```
function performTask(array) {
  _.each(array, function(elem) {
    doSomething(array[i]);
  });
}
```

15. As with any story, there is always precedent. Prior to V8, the WebKit project worked on the SquirrelFish Extreme engine that compiled JavaScript to native code. Prior to SquirrelFish was the Tamarin VM, which was developed by Mozilla based on the ActionScript VM 2 by Adobe. Even more interesting is that most JavaScript optimization techniques were pioneered by the older programming languages Self and Smalltalk.

16. Don't worry if you're not familiar with these optimization techniques. They are not important for the purposes of this book, and will therefore not be on the test. I highly encourage studying up on them, however, as they are nonetheless a fascinating topic.

17. A fun site that tracks worldwide IE6 usage is found at *http://www.ie6countdown.com*.

```
// ... some time later

performTask([1,2,3,4,5]);
```

A static optimizer could transform the body of `performTask` into the following:

```
function performTask(array) {
  for (var i = 0, l = array.length; i < l; i++) {
    doSomething(array[i]);
  }
}
```

And a sophisticated optimization tool could optimize this further by eliminating the function call altogether:

```
// ... some time later

var array123 = [1,2,3,4,5];

for (var i = 0, l = array123.length; i < l; i++) {
  doSomething(array[i]);
}
```

Finally, a really amazing static analyzer could optimize it even further into five separate calls:

```
// ... some time later

doSomething(array[1]);
doSomething(array[2]);
doSomething(array[3]);
doSomething(array[4]);
doSomething(array[5]);
```

And to top off this amazing set of optimizing transformations, you can imagine that if these calls have no effects or are never called, then the optimal transformation is:

```
// ... some time later
```

That is, if a piece of code can be determined to be "dead" (i.e., not called), then it can safely be eliminated via a process known as *code elision*. There are already program optimizers available for JavaScript that perform these types of optimizations—the primary being Google's Closure compiler. The Closure compiler is an amazing piece of engineering that compiles JavaScript into highly optimized JavaScript.[18]

There are many different ways to speed up even highly functional code bases using a combination of best practices and optimization tools. However, very often we're too

18. There are caveats that go along with using the Google Closure compiler, primary among them being that it works to its optimal efficiency given a certain style of coding. However, when it works, it works wonders, as I learned during my work on the ClojureScript compiler.

quick to consider matters of raw computation speed before we've even written a stitch of correct code. Likewise, I sometimes find my mind drifting toward speed considerations even if raw speed is not needed for the types of systems that I create. Underscore is a very popular functional programming library for JavaScript, and a great many applications do just fine with it. The same can be said for the heavyweight champion of JavaScript libraries, jQuery, which fosters many functional idioms.

Certainly there are legitimate domains for raw speed (e.g., game programming and low-latency systems). However, even in the face of such systems' execution demands, functional techniques are not guaranteed to slow things down. You would not want to use a function like `nth` in the heart of a tight rendering loop, but functional structuring in the aggregate can still yield benefits.

The first rule of my personal programming style has always been the following: Write beautiful code. I've achieved this goal to varying degrees of success throughout my career, but it's always something that I strive for. Writing beautiful code allows me to optimize another aspect of computer time: the time that I spend sitting at a desk typing on a keyboard. I find a functional style of writing code to be exceptionally beautiful if done well, and I hope that you'll agree by the time you reach the end.

The Case for Underscore

Before moving on to the meat of the book, I'd like to explain why I chose Underscore as my mode of expression. First of all, Underscore is a very nice library that offers an API that is pragmatic and nicely functional in style. It would be fruitless for me to implement, from scratch, all of the functions useful for understanding functional programming. Why implement `map` when the idea of "mappiness" is more important? That's not to say that I will not implement core functional tools in this book, but I do so with Underscore as a foundation.[19]

Second, there is a greater than zero chance that running the preceding code snippets using `Array#map` did not work. The likely reason is that in whichever environment you chose to run it might not have had the `map` method on its array implementation. What I would like to avoid, at any cost, is getting bogged down in cross-browser incompatibility issues. This noise, while extremely important, is a distraction to the larger purpose of introducing functional programming. The use of Underscore eliminates that noise almost completely![20]

19. There are other functional libraries for JavaScript that could have served just as well as a foundation for this book, including Functional JavaScript, Bilby, and even jQuery. However, my go-to choice is Underscore. Your mileage may vary.

20. Cross-browser compatibility is an issue in the use of Underscore when it defers to the underlying methods; the speed of execution is likely to be much faster than the Underscore implementation.

Finally, JavaScript by its very nature enables programmers to reinvent the wheel quite often. JavaScript itself has the perfect mix of powerful low-level constructs coupled with the absence of mid- and high-level language features. It's this odd condition that almost dares people to create language features from the lower-level parts. Language evolution will obviate the need to reinvent some of the existing wheels (e.g., module systems), but we're unlikely to see a complete elimination of the desire or need to build language features.[21] However, I believe that when available, existing high-quality libraries should be reused.[22] It would be fun to re-implement Underscore's capabilities from scratch, but it would serve neither myself (or my employer) nor you to do so.

Summary

This chapter covered a few introductory topics, starting with the motivation for learning and using JavaScript. Among the current stable of popular programming languages, few seem more poised for growth than JavaScript. Likewise, the growth potential seems almost limitless. However, JavaScript is a flawed language that needs to call on powerful techniques, discipline, or a mixture of both to be used effectively. One technique for building JavaScript applications is called "functional programming," which in a nutshell, consists of the following techniques:

- Identifying an abstraction and building a function for it
- Using existing functions to build more complex abstractions
- Passing existing functions to other functions to build even more complex abstractions

However, functions are not enough. In fact, functional programming very often works best when implemented in concert with powerful data abstractions. There is a beautiful symmetry between functional programming and data, and the next chapter dives into this symmetry more deeply.

21. This a facet of the nature of programming in my opinion.

22. I'm particularly enamored with the microjs website for discovering JavaScript libraries.