Author Picks

FREE

Designing Scalable JavaScript Applications

Chapters selected by
Emmit A. Scott, Jr.

manning

Designing Scalable JavaScript Applications

Selections by Emmit A. Scott, Jr.

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294174
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

If you've ever written a non-trivial JavaScript application, you know that creating a code base that's easy to maintain and scales well over time is no small feat. The complexity only grows as the project becomes larger. While it's impossible to plan for every change that could possibly happen over the life of a project, you can help future-proof your front-end architecture by designing software that's pliable and easily extended. This approach inherently leads to more sustainable code even in a constantly changing environment. Well-designed, extensible architecture can also help reduce development and maintenance costs, as bug fixes, enhancements, and new technologies can be incorporated more easily.

So how do you begin? Spending more time up front on the design of your application is a great start. This includes thinking about which tools and frameworks you'll use and which design patterns you'll implement.

This mini-book brings together excerpts from four different Manning titles. The chapters selected are all great starting points for understanding how to build better JavaScript applications. They'll introduce you to some fundamental concepts for creating clean, loosely coupled code, and show you how to make your development process more productive and efficient.

The role of MV frameworks*

Although you can build nearly anything from scratch with JavaScript, most of the time, starting with existing tools and frameworks is more efficient. Understanding how to take advantage of JavaScript frameworks is pivotal to building better applications. Frameworks hide the complexity normally associated with large projects and help to achieve a separation of concerns in an application's code base. They can be used with applications of any size, but are crucial for large-scale application development. The following chapter introduces the role of Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) frameworks and includes a complete working project built with three different frameworks for comparison.

Chapter 2 from *SPA Design and Architecture*
by Emmit A. Scott, Jr.

The role of MV frameworks*

This chapter covers

- An overview of UI design patterns
- An introduction to MV* in the browser
- Exposure to core MV* concepts
- Benefits of MV* libraries/frameworks
- A list of considerations when choosing a framework

Probably one of the most difficult tasks a web developer faces is creating a code base that can grow gracefully as the project grows. The larger and more complicated a project becomes, the more difficult the task. Shaping a project's code base in a way that makes troubleshooting, maintenance, and enhancements easier, not harder, is no small feat, though. This is true for even traditional web projects.

In an SPA, keeping your code segregated based on its functionality is more than just a good practice. It's critical to being able to successfully implement an application as a single page. The key to this is creating a separation of concerns within your application. Having a separation of concerns within your code base means that you're making a concerted effort to separate the various aspects of the application's code based on the responsibility it has.

Inline style — Embedded script

```
<button style="background-color: #ccc;" onClick="if(formValid && !formChanged){showConfirmation();}">
        Confirm
</button>
```

Figure 2.1 Indiscriminately interweaving JavaScript, HTML, and CSS makes your project more difficult to manage as it grows.

We can break the overall SPA into many application layers, on the server side as well as in the client. Within the browser space, we can begin our quest for creating a separation of concerns in a fundamental way by first remembering the roles of our three primary languages: HTML, CSS, and JavaScript:

- *HTML*—This is the scaffolding of your application. This code is primarily concerned with the elements that provide placeholders for content, give the UI its structure, and offer controls the user can interact with.
- *CSS*—Style sheets describe the design of the UI, giving it its look and formatting.
- *JavaScript*—In a general sense, the code in this layer represents the application's presentation logic. This layer is used to give a web application its dynamic nature, providing behavior and programmatic control over the other two layers.

We've all worked with these three languages and understand the role of each. Even so, because these three languages can easily be mixed together, your code can quickly turn into spaghetti (see figure 2.1). This can make your project extremely challenging to manage.

It's possible, however, to produce code written in a decoupled manner but still achieve the same level of interactivity between each layer. Each part of your code can be compartmentalized based on the purpose it serves.

Additionally, this compartmentalization can be extended to encompass the application's data versus its presentation to the user. Data and events can be assigned to your UI via mapping, instead of direct assignment in your business logic. The UI and the data can both be observed for changes, allowing them to stay in sync and giving your logic a means to react appropriately. So not only can the code itself be segregated based on its responsibility, but your UI's presentation and the data it represents can be disjoined. This achieves yet another level of separation in your application (see figure 2.2).

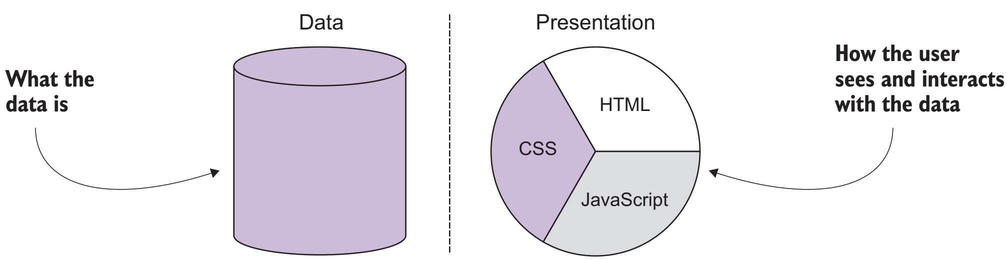


Figure 2.2 The data of your application can be separated from its representation in the UI.

Although it's entirely possible for you to create a homemade solution to manage all these layers of separation, it's probably not where you want to spend your development time. Thankfully, though, a myriad of libraries and frameworks are ideal for just such a task. If used in your application, they can play a key role in creating a successful separation of concerns by externally managing the relationship between your logic, data, and the UI. In varying degrees, they also provide many other features to assist you in building your SPA.

This chapter defines JavaScript MV*, briefly discusses its evolution from traditional UI design patterns, and discusses what these frameworks do for us. The chapter also breaks down some common MV* concepts, using three MV* frameworks to illustrate different approaches to the same objective. As mentioned in chapter 1, not everything is MV*. My focus in this book, however, is on the MV* style of frameworks.

To further demonstrate how the same concepts are prevalent in MV*, even though the approach may differ, you'll create a small but reasonably realistic online directory project with each framework. I'll abbreviate the code samples in the text so the concepts don't get lost in the code. (The code for all three versions is available in appendix A, with a complete code walk-through. And the code for each is available for download online.)

The end of the chapter includes a list of things to consider when selecting a framework that's right for you. Because all MV* implementations are different approaches to the same problem, you'll ultimately have to decide which one is the right fit for you. There's no clear right or wrong answer that fits all situations. Once you understand the role of MV* and its underlying patterns, though, you'll be able to select one that best fits your environment. After all, no one knows your situation better than you do. You know the factors that affect your project, your end users, your budget, your timelines, and your development resources.

2.1 What is MV*?

The term *MV** represents a family of browser-based frameworks that provide support for achieving a separation of concerns in your application's code base, as discussed in the introduction. These frameworks have their roots in traditional UI design patterns, but the degree to which they follow a pattern varies from implementation to implementation.

In MV*, the *M* stands for *model* and the *V* stands for *view*. Section 2.2 covers them in depth, but for this discussion let's briefly summarize each term:

- *Model*—The model typically contains data, business logic, and validation logic. Conceptually, it might represent something like a customer or a payment. The model is never concerned with how data is presented.
- *View*—The view is what the user sees and interacts with. It's a visual representation of the model's data. It can be a simple structure that relies on other parts of the framework for updates and responses to user interactions or it can contain logic, again depending on the MV* implementation.

As you'll see in section 2.1.1, traditional UI design patterns include a third component, which helps manage the relationship between the model and the view, as well their relationship with the application's user. Although most modern browser-based UI design frameworks based on MVC/MVVM have some notion of a model and a view, it's the third component that varies, in both name and the duty it performs. Therefore, people have generally settled on the wildcard (*) to represent whatever the third component might be.

Section 2.1.2 presents a lot more about MV* in the browser. First, though, let's find out a little about traditional UI design patterns, which form the roots of MV*. Knowing how we got here will help you get a better idea of why things work the way they do.

2.1.1 Traditional UI design patterns

Using architectural patterns to separate data, logic, and the resulting representation of the output is a notion that has been around for a long, long time. Central to these design patterns is the idea that an application's code is easier to design, develop, and maintain if it's segmented based on the type of responsibility each layer has.

This section details the three patterns that have had the most influence on client-side approaches: Model-View-Controller (MVC), Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM). After a proper introduction to these design patterns, you'll see in section 2.1.2 how they relate to the MV* frameworks we see in the browser today.

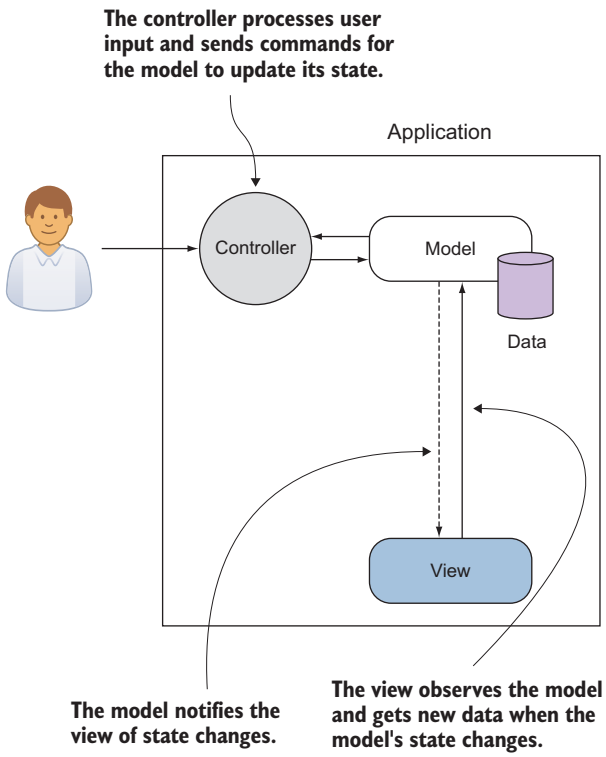


Figure 2.3 The MVC design pattern has been used for many years in the development of graphical user interfaces.

MODEL-VIEW-CONTROLLER

Model-View-Controller (MVC) is one of the oldest patterns to try to separate data, logic, and presentation. MVC was proposed by Trygve Reenskaug and later implemented in the Smalltalk programming language in the 1970s.

MVC was instrumental in the design of graphical user interfaces then and still is today. Since its inception, it and its variants have become common design patterns for all types of software development. The MVC pattern includes the model, the view, and a controller (see figure 2.3):

- *Controller*—The controller is the entry point for the application, receiving signals from controls in the UI. It also contains the logic that processes the user input and sends commands to the model to update its state based on the input received.

The interactions with the controller set off a chain of events that eventually lead to an update of the view. The view is aware of the model in this pattern and is updated when changes are observed.

MODEL-VIEW-PRESENTER

In 1996, a subsidiary of IBM called Taligent came up with a variation of MVC called *Model-View-Presenter*, or *MVP*. The idea behind this pattern was to further decouple the model from the other two components of MVC. Under MVP, a controller-like object and the view would jointly represent the user interface, or *presentation*. The model would continue to represent data management. As noted in figure 2.4, in MVP, there's no controller acting as a gatekeeper. Each view is backed by a component called a presenter:

- *Presenter*—The presenter contains the view's presentation logic. The view merely responds to user interactions by delegating responsibility to the presenter. The presenter has direct access to the model for any necessary changes and propagates data changes back to the view. In this way, it acts as a "middleman" between the model and the view.

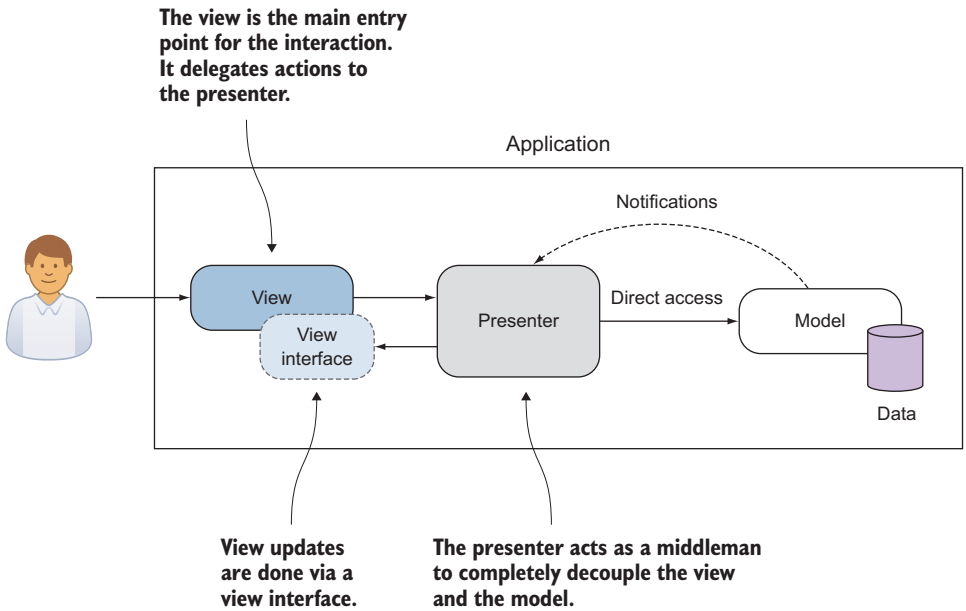


Figure 2.4 MVP is a variation of MVC. With this pattern, the view is the entry point, but its logic is in the presenter.

The presenter takes on the task of keeping the view and model updated. Having an object in the middle allows the view and model to have more-focused responsibilities.

MODEL-VIEW-VIEWMODEL

Model-View-ViewModel (MVVM) was proposed by John Gossman in 2005 as a way to simplify and standardize the process of creating user interfaces for use with Microsoft's Windows Presentation Foundation (WPF). It's another design pattern that emerged to try to organize the code associated with the UI into something sensible and manageable, while still keeping the various components of the process separate.

As in MVP, the view itself is the point of entry. Also like MVP, this model has an object that sits between the model and the view (see figure 2.5). The third component in this pattern is called the ViewModel:

- *ViewModel*—The ViewModel is a model or representation of the view in code, in addition to being the middleman between the model and the view. Anything needed to define and manage the view is contained within the ViewModel. This includes data properties as well as presentation logic. Each data point in the model that needs to be reflected in the view is mapped to a matching property in the ViewModel. Like a presenter in MVP, each view is backed by a ViewModel. It's aware of changes in both the view and the model and keeps the two in sync.

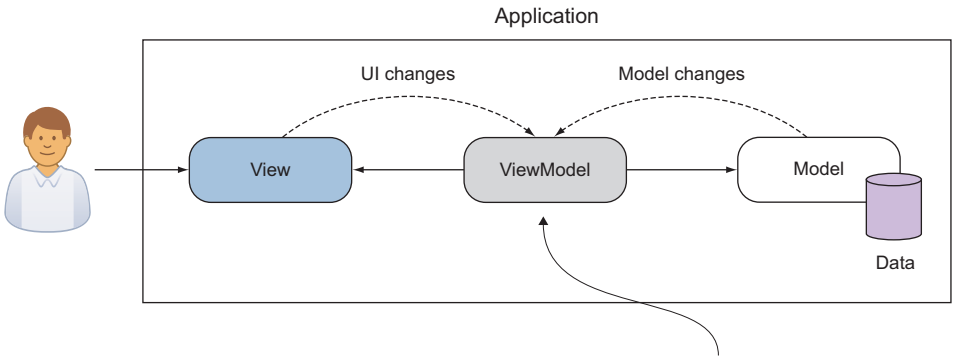


Figure 2.5 In MVVM, the ViewModel is aware of changes in both the model and the view and keeps the two in sync.

Now that you know a little about traditional UI design patterns, you can better understand browser-side MV* approaches. Let's fast-forward, then, and talk about the MV* we see in the browser.

2.1.2 MV* and the browser environment

Just like application code running on the server or natively as a desktop application, code running in the browser can benefit from using good architectural design patterns. In recent years, many frameworks have cropped up, aiming to fulfill this need.

Most are based on MVC, MVP, or MVVM to some degree. The browser is a different sort of environment, though, and we're dealing with three languages at once (JavaScript, HTML, and CSS). Therefore, it's difficult to perfectly match a browser-side MV* framework with a design pattern. Trying to pigeonhole them into one category or another is, in most cases, a fruitless undertaking. Design patterns should be malleable strategies, not inflexible directives.

One of the reasons why the term *MV** sprang up in the first place is that it's often hard to nail down what the third concept in the framework is. The term represents sort of a compromise, to cease the endless disputes about whether particular frameworks are more this pattern or more that pattern.

The remnants of the traditional patterns are there but are loosely interpreted. Each one has some form of the data model, whether it's in the form of a *POJO* (plain old JavaScript object) or some model structure dictated by the implementation. Each also has some notion of a view. The third cog in the machine might be a little more elusive, though. The framework might employ an explicit controller, presenter, or ViewModel. But it might have some sort of hybrid or not have the third part at all!

Derick Bailey, creator of Marionette.js for Backbone.js, put things rather eloquently in one of his online posts, titled "Backbone.js Is Not an MVC Framework":

> *Ultimately, trying to cram Backbone into a pattern language that doesn't fit is a bad idea. We end up in useless arguments and overblown, wordy blog posts like this one, because no one can seem to agree on which cookie-cutter pattern name something fits into. Backbone, in my opinion, is not MVC. It's also not MVP, nor is it MVVM (like Knockout.js) or any other specific, well-known name. It takes bits and pieces from different flavors of the MV* family and it creates a very flexible library of tools that we can use to create amazing websites. So, I say we toss MVC/MVP/MVVM out the window and just call it part of the MV* family.*
>
> Source: http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/

Many other people share this same viewpoint about the fruitlessness of trying to one-for-one match today's MV* with a traditional design pattern. The idea of the usefulness of the framework taking priority over its categorization gained even more steam when the AngularJS team weighed in with a similar conclusion about their framework. Igor Minar (from the AngularJS team) famously blogged that developers will argue endlessly about how to categorize a particular MV* framework. He went on to state that AngularJS started out more like MVC, but over time it has become a little more like MVVM. In truth, it's a little like both. In this same blog entry, he proposes the term *MVW*, which has since stuck:

> *I'd rather see developers build kick-ass apps that are well-designed and follow separation of concerns than see them waste time arguing about MV* nonsense. And for this reason, I hereby declare AngularJS to be MVW framework—Model-View-Whatever. Where Whatever stands for whatever works for you.*
>
> Source: https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV

Knowing that most MV* implementations only loosely base their design on the original pattern helps us remember that it's not so important to try to brand the framework as one pattern or another.

2.2 Common MV* concepts

Now that you know what MV* is, let's go over a few common concepts that are frequently found, no matter the implementation. In the examples for each concept, you'll quickly begin to see that even though the syntax and approach may vary between frameworks, the ideas are the same. Before we begin, let's take a moment to review at a high level the concepts covered in this section:

- *Models*—Models represent the data of our application. They contain properties and possibly logic to access and manage the data, including validation. The model often contains business logic as well.
- *Views*—Views are what the user sees and interacts with and are where models are visually represented. In some MV* implementations, views may also contain presentation logic.
- *Templates*—Templates are the reusable building blocks for views when dynamic content is needed. They contain placeholders for data, as well as other instructions for how content in the template should be rendered. One or more templates will be used to create a view in an SPA.
- *Binding*—This term describes the process of associating the data from a model with an element in a template. Some MV* implementations also provide other types of binding, such as bindings for events and CSS styles.

Figure 2.6 gives a big-picture view of how these concepts relate to each other in an SPA. These concepts are probably the least common denominator in building SPAs. Other features, such as routing (covered in chapter 4), are also common (and necessary) but may not be provided universally. Not to worry, though. I cover many of the other concepts later in the book. We just need a sound foundation to begin with.

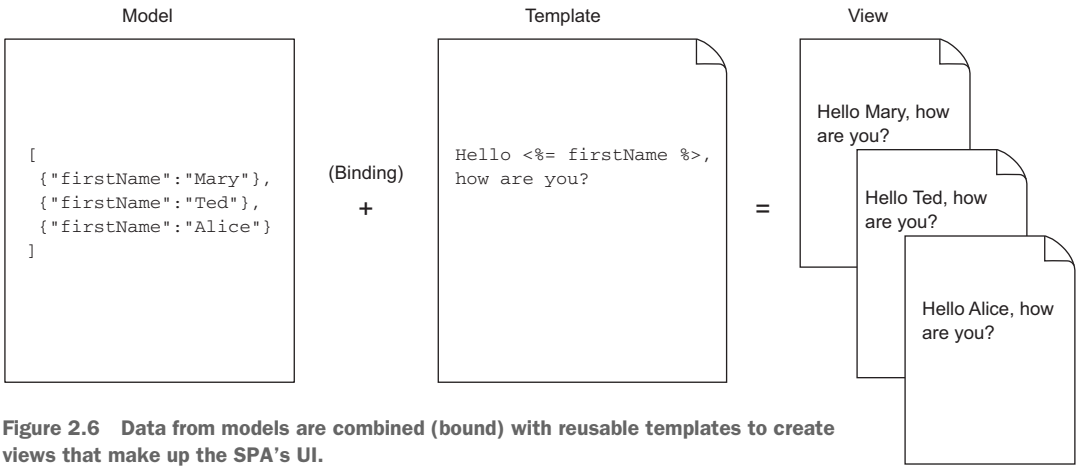


Figure 2.6 Data from models are combined (bound) with reusable templates to create views that make up the SPA's UI.

2.2.1 Meet the frameworks

Because we're using three frameworks for illustration in this section, some introductions are in order. Each represents a slightly different approach to these basic MV* concepts. Seeing the different approaches, though, should give you a broader perspective ultimately. The three frameworks are as follows:

URL: http://backbonejs.org

Description: As mentioned before, Backbone.js doesn't perfectly fit a traditional design pattern but could be described as being somewhere between MVC and MVP. Backbone.js is code driven. Models and views are created programmatically, using JavaScript code in this framework, by extending Backbone.js objects. By extending core objects, you automatically inherit a lot of built-in functionality. The framework also provides other out-of-the-box features to make routine tasks easier. Backbone.js doesn't provide everything you'll need in your SPA, though, so you must fill certain gaps using other libraries or frameworks.

URL: http://knockoutjs.com

Description: Knockout may not perfectly fit with the original MVVM definition, but it's fairly close. In this framework, the model is *any* source of data, not an explicit object structure prescribed by the framework. Views and templates are created with plain HTML. The ViewModels that map model data to UI elements and provide views with behavior are created programmatically using JavaScript code, but most everything else is done *declaratively* by adding custom attributes to the HTML. Knockout is mainly concerned with making the binding process clean and easy. Though this makes the framework small and superbly focused, it leaves you to look to other frameworks and libraries for all other SPA requirements.

URL: https://angularjs.org

Description: AngularJS humbly describes itself as the "Superheroic JavaScript MVW Framework." The creators of AngularJS designed it to be a one-stop-shopping kind of framework. Most, if not all, of your SPA needs are covered by this framework. AngularJS mixes and matches concepts that its creators liked from traditional patterns, as well as from other popular frameworks, to come up with a nicely balanced palette of out-of-the-box features. Part of your work in this framework will be done programmatically via JavaScript code, and part will be done declaratively using custom HTML attributes.

2.2.2 Meet our MV* project

To help illustrate our list of common concepts, you'll create a simple online employee directory. You'll create it three ways, using each of the frameworks previously

described. Later in the book, you'll learn about more-advanced topics, such as routing and server transactions. For now, you'll stick to the basics for this project. Our example, though somewhat contrived, still covers basic CRUD operations over a list. That should be sufficiently challenging for an introduction.

Let's go over our objectives for this example:

- Create a simple SPA to enter employee information.
- Build an easy-to-use UI for entering each employee's first name, last name, title, email, and phone.
- Keep track of each entry as part of a list, with the screen split between the entry form on the left and the directory's entry list on the right.
- Have two buttons on the entry side of the SPA: one to add a new entry and one to clear the form.
- Have one button next to each entry to remove the entry from the list.
- Have indicators next to each entry field to denote whether the field's entry requirement has been met. (Each indicator should update as the user types.)

Now that you've reviewed the objectives, take a look at the screenshot of the final product (see figure 2.7). The application will look and behave the same for each MV* framework used.

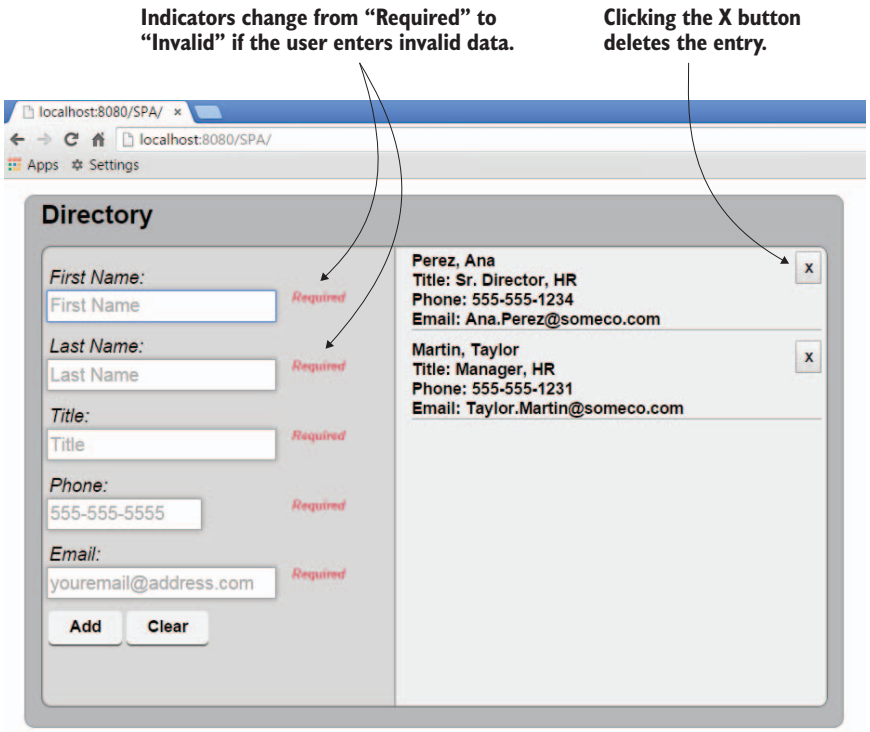


Figure 2.7 Screen capture of the online directory. The user enters information in the form on the left. Valid entries appear in a list on the right.

Throughout our discussion of MV* concepts, I'll refer to this example. I'll also talk about how different philosophies by the framework creators affect the type of code you'll create. Although only certain parts of the code will be used for illustration, all of the code for each MV* version is in appendix A and available for download.

2.2.3 Models

You know from our discussion of patterns that models often contain business logic and validation. They also represent the data in your application. The data they contain shouldn't be a motley crew of unrelated information, though. They're called *models* because they model a real-life entity that's important to the application's logic.

For example, if you were building an online reservation system for a hotel, your models might include the hotel, a room, an agent, a customer, the reservation, amenities, notes, invoices, receipts, and payments. What about a web-based application for teachers? You'd need to have data representing the school, its teachers, the students, courses, and grades, at a minimum. Each model in the application would represent a real-world object. Consequently, the larger and more complex the system, the more types of models you'll have.

Let's see what a model's data would look like for our online employee directory. Remember that models mirror things in the real world. They contain not only data but behavior as well. In this case, you're going to model a directory listing. I'll keep it extremely simple to make sure the concept doesn't get buried in too much code. Here's the information we're going to keep track of in each model:

- First name
- Last name
- Title
- Email address
- Phone number

The employee list inside the directory would be a collection (array) of these models. Figure 2.8 illustrates what this would look like from a conceptual standpoint.

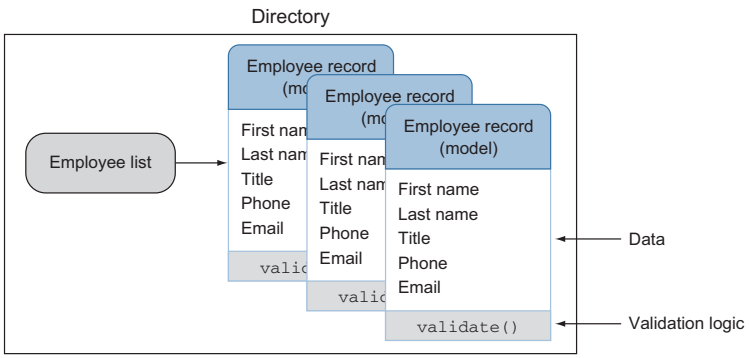


Figure 2.8 The application's data in our online directory project is just an array of employee models. Each model is an object that contains the employee information we'll see onscreen.

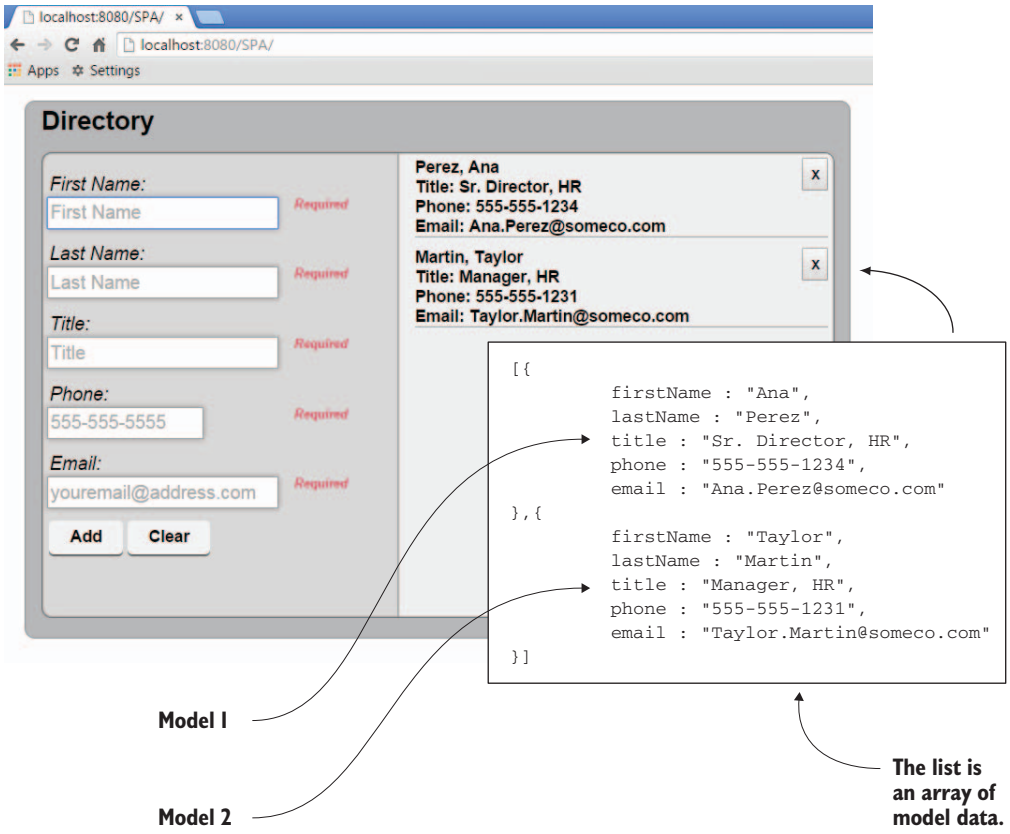


Figure 2.9 Screen capture of the online directory. Here you see that two instances of the employee model have been added to the list.

To help you visualize what the models will look like when they're added to the collection and rendered in the view, you can look at the screenshot of the employee directory application again (see figure 2.9). Remember, each model in the list is an object in an array. In this version of the screenshot, I've superimposed a snapshot of the data inside each model that has been added to our list. This will help you see the models in action.

Now that you have a mental picture of how the employee models will be used, let's talk about how a model is defined in each type of framework. How you create a model in an MV* application varies, depending on the framework you're using. As I mentioned, the framework might not be an exact match for one of the traditional design patterns, but the implementation will certainly be influenced by the pattern.

IMPLIED MODELS

In some MV* implementations, the model is just the data itself, not an explicit structure prescribed by the framework. This data can be from any source, including POJOs and HTML form controls in the UI itself. There are no restrictions on what you can use

for the source data when the model is implicit. This is the case for both Knockout and AngularJS.

For example, in our fictitious employee directory, we need the data for the employee model to come from the entry form the user fills out. So instead of creating a JavaScript object or getting JSON from the server, we need to capture data directly from the `INPUT` fields of our HTML form when creating each entry in the directory's list.

AngularJS provides an easy shortcut for this. If you need your model data to come from an `INPUT` field, you can add a custom attribute called `ng-model` to *each* field (see listing 2.1). The attribute declares that model data is sourced from the HTML form element where the attribute is placed. The attribute magically establishes the `formEntry` model if it doesn't already exist and gives it a property called `firstName`. Then it ties `formEntry.firstName` to this `INPUT` field.

Listing 2.1 AngularJS model

```
<input id="firstName" name="firstName" type="text"
ng-model="formEntry.firstName"
required
placeholder="First Name"/>
```

ng-model is AngularJS's custom attribute

Once established, the model is readily available in your JavaScript code. One of the many benefits of using an MV* framework is keeping the complex, boilerplate code that marries the data and UI external to our application's logic. This one attribute is a great example.

In Knockout, the model is again implied, not explicitly declared (see listing 2.2). In this framework, you add custom attributes to *each* `INPUT` field just as we did in the AngularJS version. This time, the attribute is called `data-bind`.

With Knockout (in true MVVM fashion), the attribute ties the `INPUT` field with a matching property in the ViewModel. In turn, our JavaScript code gains access to the field through the ViewModel.

Listing 2.2 Knockout model

```
<input id="firstName" name="firstName" type="text"
data-bind="hasFocus: isFocused,
value: entry.firstName,
valueUpdate: 'afterkeydown'"
placeholder="First Name" />
```

data-bind is Knockout's custom attribute

A matching property from a ViewModel you'll define in code

With both AngularJS and Knockout, your model could have been any data source. Because you're working from an entry form, that's where you needed your model's data to come from. In both cases, no model object was explicitly defined. Instead, each framework provided a custom attribute you could add to the HTML to establish the entry form as the source of the model's data. Now let's see how to create a model in Backbone.js, where models are explicitly defined in code.

EXPLICIT MODELS

In MV* implementations where an explicitly declared model is required by the framework, the model is created as a JavaScript object. Backbone.js is a prime example of this. Backbone.js models can have logic in addition to data, such as validation, default data, and custom functions. You also inherit a lot of functionality. Just by extending the framework's model to create your own, you automatically receive a wide variety of base functionality without even writing code.

The ability to immediately inherit a lot of functionality makes creating a model in these types of frameworks powerful and flexible. For example, in Backbone.js you can create a bare-bones model in a single line of code:

```
var EmployeeRecord = Backbone.Model.extend({});
```

To use the object, you create a new instance of it and call any functions it has available out of the box. With this single declaration, your model right away has a variety of built-in behaviors such as validation, functions to execute RESTful services, and much more. See the online documentation for the full list (http://backbonejs.org).

It's equally as easy to assign properties to a Backbone.js model. To create a new property called `firstName` and set its value to `Emmit`, you can either pass in `{firstName : "Emmit"}` to the object's constructor or use the model's built-in `set` method:

```
var employee = new EmployeeRecord({});
employee.set({firstName : "Emmit"});
```

The following listing illustrates the Backbone.js version of the employee model for our online directory. The validation needed for the directory example makes the source for the model quite verbose.

Listing 2.3 Backbone.js model

```
var validators = {
   "*": [ {
      expr: /\S/,
      message: "Required"
   } ],
   "phone": [ {
      expr: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/,
      message: "Invalid"
   } ],
   "email": [ {
      expr: /^[a-z0-9!#$%&'*+\/=?^_`{|}~-]+(?:\.
      [a-z0-9!#$%&'*+\/=?^_`{|}~-]+)*@
      (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
      (?:[a-z0-9-]*[a-z0-9])?$/i,
      message: "Invalid"
   } ]
};

function validateField(value, key) {
   var rules = validators["*"].concat(validators[key] || []);
```

Define validation for data being set on the model's properties

Keep track of specific errors

```
    var broken =
    _.find(rules,function(rule) {return !rule.expr.test(value);});

    return broken ? {"attr":key,"error":broken.message} : null;
}

var EmployeeRecord = Backbone.Model.extend({
    validate: function(attrs) {
        var validated = _.mapObject(attrs, validateField);
        var attrsInError = _.compact(_.values(validated));
        return attrsInError.length ? attrsInError : null;
    },
    sync: function(method, model, options) {
        options.success();
    }
});
```

Create the model

There seems to be a lot more going on than there is. We inherit the power of the Backbone.js model just by extending its base object. But the framework leaves it up to you as to how you want to validate the data. It gives you a couple of hooks with `validate(attrs, options)`, and you can fill in the rest however you want.

2.2.4 Bindings

The term *binding* is another concept you should understand if you plan on using an MV* framework. This term is used frequently when talking about UI development. In plain English, it means to tie or connect two things together. In UI development, whether we're talking about desktop programming with a language like .NET or web development with MV*, we mean linking a UI element in the view (such as a user input control) to something in our code (for example, a model's data).

It doesn't have to be just data, though. Different libraries and frameworks offer different types of bindings. Styles, attributes, and events such as `click` are just a sampling of what can be bound to the UI. The binding types that are available vary, depending on the framework. You'll look at the code for a few approaches in this section, just to illustrate.

How exactly do we declare a binding in our application? MV* frameworks make binding something in our code to an element in the UI simple. Understanding how to declare a binding starts with getting to know the syntax.

BINDING SYNTAX

Binding syntax comes in two flavors:

- *Expressions*, which are special characters that wrap/delimit the bound item
- HTML *attributes* (called *directives* in AngularJS or *bindings* in Knockout)

With both types, the binding syntax is freely mixed with the HTML of the template. Table 2.1 lists a few examples of the binding syntax used by some popular libraries/frameworks. This is by no means an exhaustive list, but it should give you a general idea.

Keep in mind, also, that the table is using simple text bindings to illustrate syntax styles. As noted previously, things other than data, such as events and CSS styles, may also be supported. See the documentation for each library/framework to see the complete list of bindings supported and additional usage instructions.

Table 2.1 Binding is in the form of either an attribute or an expression. AngularJS supports both styles to some extent.

Framework/library	Type	Example
Knockout http://knockout.com	Attribute	`data-bind="text: firstName"`
AngularJS (type 1) https://angularjs.org	Attribute	`ng-bind="firstName"`
AngularJS (type 2)	Expression	`{{ firstName }}`
Mustache http://mustache.github.io	Expression	`{{ firstName }}`
Handlebars http://handlebarsjs.com	Expression	`{{ firstName }}`
Underscore.js (default) http://underscorejs.org	Expression	`<%= firstName %>`

After you look at the documentation of the framework or library to get a feel for the syntax, the next thing to understand about binding is the directional flow of data in the binding.

BINDING DIRECTION

Binding something in our code to a visual element in the view can be bidirectional, single-directional, or a one-time binding. The type of binding relationship is also established via the MV* framework.

TWO-WAY BINDING

In bidirectional, or two-way, binding, after the binding link is established, changes on either end cause updates on the opposite side. This keeps the two sides in sync. In a web application, two-way binding is associated with UI controls, like those in a form, that support user input.

Knockout is a great library to illustrate the concept of two-way binding. Binding is, after all, this library's main purpose. As you saw previously, creating a binding is as easy as typing a custom attribute called `data-bind` right in the HTML. The `data-bind` attribute tells Knockout that something in the UI is going to be bound to a property in a ViewModel. In the following example, we're binding the *value* of an `INPUT` control to a ViewModel property called `firstName`:

```
<input data-bind="value: firstName" />
```

For the other half of the two-way binding relationship, you tell Knockout you want the property to be observed for any changes by wrapping its data in a Knockout observable object. (Remember observables from the Observer pattern?)

```
var myViewModelObject = {
   firstName : ko.observable("Emmit")
};
```

Because of the two-way binding established by this small amount of code, these two items will stay in sync automatically.

Binding is just as easy, or more so, with AngularJS. You already saw AngularJS's two-way binding in action during our discussion of models. You add the attribute `ng-model` to the HTML of the `INPUT` tag:

```
<input ng-model="firstName" />
```

On the JavaScript code side, you have a `$scope` object instead of a ViewModel. Scopes are similar in that they sit between the view and our JavaScript code and give us access to the model.

One nice thing about this framework is that AngularJS's magic automates a lot of the two-way setup. First, the `$scope` object automatically monitors models for changes. Second, you don't even have to create the `$scope` object; AngularJS will hand it to you, if you ask for it. Then, in your code, you can refer to the property via the `$scope` object like this:

```
$scope.firstName
```

That's it. Now both the `INPUT` field and the property are bidirectionally bound. Changes on either side affect the other. Pretty easy, huh?

You've seen the approaches to two-way binding from two different frameworks. Even so, the concept remains the same in both. Now let's take a quick look at binding in a single direction.

ONE-WAY BINDING

When binding is single-directional, or one-way, changes in the state of the source affect the target but not the other way around. This type of binding is normally associated with HTML elements that don't require any input from the user, such as a `DIV` or `SPAN` tag. With these types of elements, you're interested in its text, not its value. You still access the data on the JavaScript side in the same manner, but in the template you choose the attribute specifically for one-way text binding.

With Knockout, you change the word *value* to *text*:

```
<span data-bind="text: firstName"></span>
```

In AngularJS, the attribute itself changes from `ng-model` to `ng-bind`:

```
<span ng-bind="firstName"></span>
```

Once again, you can see that the binding concept remains the same even though you're looking at different MV* frameworks.

TIP Knockout provides an additional way to make even bindings for user input one-way, in case you need that behavior. You just remove the observable "wrapper" from the ViewModel property like this: `firstName : "Emmit"`.

You might be wondering at this point, why bother with the one-way types? Why not use two-way binding always? Well, usually something as magical as automatic, two-way binding comes at a cost. Two-way binding has slightly more overhead. No need to panic and avoid it, though. For most views, this overhead is negligible. But if you have a ton of bindings throughout your application, you should use any means to save on overhead.

If your view receives input from the user, and you need the data and view to stay constantly in sync, use two-way binding. When you have read-only UI elements, use one-way binding. One-way will keep the view updated when the model changes but doesn't bother with trying to monitor the view side, because the element is read-only.

ONE-TIME BINDING

One-time binding is a type of one-way binding that happens only once. Nothing is automatically observed for changes. No subsequent updates occur if the source changes or the target changes.

With one-time binding, after the template and the data are combined and rendered as the view, the process is done. If new changes need to be applied to the view, the entire process starts over. The previous view is destroyed, and the new data is combined with the same template to generate the view anew.

I've saved Backbone.js for this section. The typical approach for rendering templates when using Backbone.js is through one-time binding (though some Backbone-compatible libraries and plugins offer the other two types). With AngularJS and Knockout, after the bindings are established, they're reused. In Backbone.js, the general idea is that when new data is needed, the view is destroyed (with the bindings) and re-created. I'll talk more specifically about templates in the next section.

NOTE Backbone.js doesn't have templating/binding capabilities built in but instead lets you pick the outside library of your choice for the task. Its default is the utility library Underscore.js.

To recap the types of bindings we discussed, consider table 2.2.

Table 2.2 Bindings can be two-way, one-way, or one-time.

Binding type	Behavior
Two-way	Bidirectional—keeps data and view constantly in sync.
One-way	Single-directional, or one-way—changes in the state of the source affect the target, but not the other way around.
One-time	One-way—occurs only once at render time, from model to view.

In the next section, you'll see binding in action with template examples from our online employee directory.

2.2.5 Templates

A *template* is a section of HTML that acts as a pattern for how our view is rendered. This pattern can additionally contain various types of bindings and other instructions that dictate how the template and its model data get processed. A template is also reusable, like a stencil.

One or more templates are used to create a view, with complex views often having multiple rendered templates on the screen at the same time. The part of the MV* framework, whether built in or via an outside library, that marries the template and the model's data is generally referred to as a *template engine*. Figure 2.10 illustrates the marriage of data from the employee directory form (our model) and a template to arrive at what the user sees onscreen.

You should now be able to recognize bindings when you see them, so let's take a look at some real examples of templates and their bindings taken from our MV* project. I've highlighted the bindings so you can easily see which part of the template is the binding and which part is just the HTML.

WHAT TEMPLATES LOOK LIKE

One thing all templates have in common is that they represent some part of our view. What this means for you as a developer is that, apart from the binding syntax, views

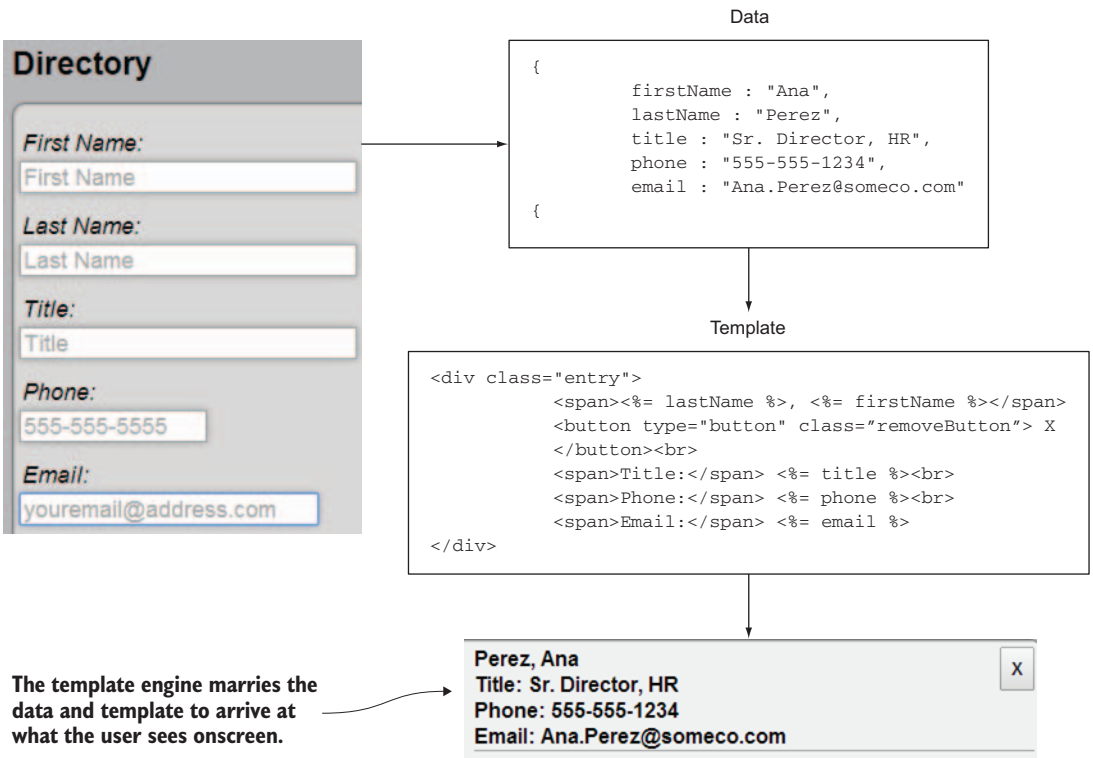


Figure 2.10 The fully rendered template, created by a template engine

are just HTML. This also means that if you have a web designer on the team, views can be constructed by the designer as well.

The following listing shows our Knockout template. Notice that it has custom attributes, but apart from that it's normal HTML.

Listing 2.4 Knockout template

```
<li class="entry">
   <button type="button" class="remove-entry"
     data-bind="click: removeEntry">&#9587;</button>

   <span data-bind="text: $data.lastName"></span>,
   <span data-bind="text: $data.firstName"></span>
   <br />

   <span>Title:</span>
   <span data-bind="text: $data.title"></span>
   <br />

   <span>Phone:</span>
   <span data-bind="text: $data.phone"></span>
   <br />

   <span>Email:</span>
   <span data-bind="text: $data.email"></span>
</li>
```

This click binding removes the entry

Data bindings link our form fields to the ViewModel's data

Do you remember from our discussion of binding syntax that with AngularJS we could use either expressions or attributes? For the online directory project, we're using expressions just to demonstrate those (see the following listing). You can use `ng-bind` if you prefer the attribute style.

Listing 2.5 AngularJS template

```
<li class="entry"  ng-repeat="entry in entries">

   <button type="button" class="remove-entry"
   ng-click="removeEntry(entry)">&#9587;</button>

   {{entry.lastName}}, {{entry.firstName}}<br />

   Title: {{entry.title}}<br />
   Phone: {{entry.phone}}<br />
   Email: {{entry.email}}

</li>
```

Stamps out this template for every entry object in the list of entries

Calls a function, passing the current entry object

What's neat about templates in Backbone.js is that you aren't confined to a particular template syntax, because there's no built-in template library. Backbone.js allows you to use the template engine of your choice. In our directory project, we're using the default, which is Underscore.js (see the following listing). Underscore.js has a default

delimiter of `<%= %>` for its expressions, but the delimiters can be anything you want (including `{{ }}`).

Listing 2.6 Backbone.js template using the Underscore.js template library

```
<button type="button" class="remove-entry">&#9587;</button>
<%= lastName %>, <%= firstName %><br />
Title: <%= title %><br />
Phone: <%= phone %><br />
Email: <%= email %>
```

We're sticking with the default Underscore.js as the template engine, but we could substitute a different one

These examples provide a good introduction to templates and the various binding styles that are used. One thing we haven't talked about yet, though, is how the process gets triggered.

TEMPLATE RENDERING

In AngularJS, the rendering of a template happens automatically, as soon as the application starts. AngularJS searches through the DOM for its custom attributes, including those for binding templates.

In other frameworks, this step isn't difficult but may be a little more explicit. Knockout, for example, requires a one-liner in your JavaScript code for each ViewModel to activate the bindings:

```
knockout.applyBindings(myViewModel, $("#someElement")[0])
```

Knockout has a special function you call, `applyBindings`, which renders the template with the model data supplied by the ViewModel. The first parameter is the ViewModel itself. The second is the place in the DOM you want Knockout to start looking for bindings for the given ViewModel. The second parameter is optional, but for efficiency you should use it in order to confine the binding process to a particular parent element.

> **TIP** `$("#someElement")[0]` is the way in jQuery to access the underlying DOM object referenced in its selector, because jQuery doesn't know how many elements will be a match for a given selector. You can also use the JavaScript `document.getElementById("someElement")` method as the second parameter.

In Backbone.js, rendering the template is a bit more of a manual process. The framework provides `template` and `render` as hooks for the external template engine of your choice. To render the template to the screen, you have to run it through the compile-and-render process. You'll see the view in its entirety in the next section, but for now let's focus on the rendering of the template, as shown in the following listing.

Listing 2.7 Backbone.js template compile and render

```
template : _.template(templateHTML),
render : function() {
    var modelAsJSON = this.model.toJSON();
```

Compiles the template into a reusable function

Translates the model data to a JSON string

Marries the data and HTML

```
    var renderedHTML = this.template(modelDataAsJSON);
    this.$el.html(renderedHTML);
    return this;
}
```

Replaces the element's content with the rendered HTML

The preceding code seems quite verbose when compared to the other two frameworks. The ability to choose whichever template engine you want is a great trade-off, though.

You now know what templates are and how to create them. The lingering question you might have is where do you keep their source code?

WHERE TO KEEP TEMPLATES

Templates can either be included in the initial download of your SPA (*inline*) or downloaded on demand as external *partials* (or *fragments*).

INLINE TEMPLATES

If your template uses the expression style of binding syntax and isn't downloaded on demand, you'll need to place it inside `SCRIPT` tags. You use `SCRIPT` tags to avoid accidentally showing the user the binding code before the render process happens. The browser won't try to display code within the `SCRIPT` tags.

To prevent the browser from trying to execute the script as JavaScript code, you'll need to give the `SCRIPT` tag's Multipurpose Internet Mail Extensions (MIME) type something other than `text/javascript` or `application/javascript`, as shown in the following listing.

Listing 2.8 Inline template

```
<script type="text/template" id="myTemplate">
    Hello, <%= firstName %>, how are you?
</script>
```

Wrapping a template in SCRIPT tags hides its source. Non-JavaScript MIME types aren't treated as JavaScript code.

If your inline template uses attributes for its binding syntax, there's nothing else special you need to do. `SCRIPT` tags aren't needed. Also, because attributes aren't displayed anyway, there's no chance the user will accidently see the bindings before the view is rendered.

TEMPLATE PARTIALS

If you download the template on demand, there's no need for `SCRIPT` tags, even if you're using expressions. The dynamically fetched template can be used directly by the template engine, which avoids the issue altogether.

As noted previously, these on-demand templates are sometimes referred to as *partials* or *fragments*. They're not part of the initial HTML document that's loaded with your application. Instead, they're fetched as snippets of source code directly from the server at runtime.

Now that you have a good idea about models, binding, and templates, you need to finally see how they culminate into the view the user sees and interacts with. In the next section, you'll look at how MV* frameworks approach views.

2.2.6 Views

As you saw during our discussion of templates, frameworks such as Knockout and AngularJS use declarative bindings in their templates, usually in the form of special attributes added to HTML elements. In these frameworks, the templates and views are pretty much the same thing. Thus, when composing views in these frameworks, you need to decide only whether the templates will be kept inline or downloaded on demand. This is more of a design issue.

In a code-driven framework like Backbone.js, the approach is to programmatically create the view. The following listing is an example from our directory application of a Backbone.js view. Don't worry if you don't understand everything in the example right now. When you're ready, appendix A contains a complete walk-through of the code.

Listing 2.9 Backbone.js view example

```
var Employee = Backbone.View.extend({
   tagName: "li",
   template: _.template(templateHTML),
   render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      return this;
   },
   events: {
      "click .remove-entry": "removeEntry"
   },
   removeEntry: function() {
      this.model.destroy();
      this.remove();
   }
});
```

Define the element type

Compile the template

Extend the built-in Backbone.js view object

Render the view and attach it to the DOM

Define the behavior for the click event

Perform some cleanup anytime the view is removed

Backbone.js allows you to define traits for your view, such as its CSS class name and the type of element it will be. Additionally, you're given the freedom to define key milestones in the life of the view, such as its rendering and removal, in any manner you wish.

The discussion on views rounds out our conversation of basic MV* concepts. It's great to understand the concepts, but what does an MV* framework do for us? In the next section, I'll discuss how using an MV* framework can make our lives as SPA developers much easier and our code much cleaner.

2.3 Why use an MV* framework?

Deciding to use any external software in your project shouldn't be taken lightly. You are, after all, introducing a dependency. That being said, however, when its benefits exceed the costs, a new dependency is worth considering. This section presents some of the key benefits of using MV* frameworks.

2.3.1 Separation of concerns

As mentioned previously, MV* frameworks provide a means to segregate JavaScript objects into their basic roles based on their underlying design pattern or patterns. Each part of the code can be focused on a particular responsibility to the application.

This overarching concept of separation of concerns helps us design objects with a particular purpose. Models can be dedicated to data, views can be dedicated to the presentation of data, and components such as controllers, presenters, and ViewModels can keep these two parts communicating with one another without being joined at the hip. The more dedicated an object is to a singular purpose, the easier it is to code, test, and update after it's in production.

MV* frameworks also reduce the tendency to write spaghetti code by providing framework elements that require us to write code in a particular way to facilitate loose coupling. This keeps our HTML as clean as possible by removing embedded JavaScript and CSS code. It also keeps our JavaScript free from deep coupling with DOM elements.

Here's a classic AngularJS example demonstrating how MV* can make our code cleaner. Figure 2.11 shows a `SPAN` tag (right) mirroring what's being typed into an `INPUT` field (left).

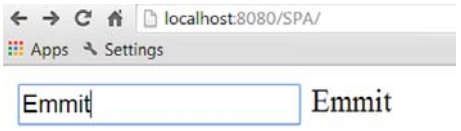


Figure 2.11 In this example, a `SPAN` tag's contents are being updated dynamically as the user types into an `INPUT` field.

Let's create the example from figure 2.11, first written as spaghetti code and then with AngularJS. Listing 2.10 is the source code, written as tightly coupled HTML and JavaScript. *Tightly coupling* means making direct references or calls from one function or component to another. This joins them at the hip, so to speak.

Writing code this way works, but it can prove to be difficult to read and a pain to update later. If an entire single-page application were written like this, you can imagine how monumentally difficult it would be to maintain.

Listing 2.10 Tightly coupled HTML and JavaScript

```
<html>
<body>
     <input id="name"
     onKeyUp="document.getElementById('output').innerHTML
     = document.getElementById('name').value">
     <span id="output"></span>
</body>
</html>
```

Don't do this!

WARNING The code in listing 2.10 dynamically updates the `SPAN` as you type but is hard to read and difficult to maintain.

Now, in the following listing, we use the AngularJS framework's ability to abstract away much of the boilerplate code needed to achieve the same results.

Listing 2.11 AngularJS example

```
<html>
<body ng-app>
    <input ng-model="name">
    <span>{{name}}</span>
    <script src="js/thirdParty/angular.min.js"></script>
</body>
</html>
```

The AngularJS framework marries the INPUT field with the SPAN tag, removing the need to put JavaScript code in your HTML

Believe it or not, that small bit of code is all that's needed. Of course, this is a contrived example. Even using a framework as powerful as AngularJS, complicated applications will have complicated logic. For all the reasons I just mentioned, though, you'll spend more of your time writing business logic, not the low-level, routine plumbing.

2.3.2 Routine tasks simplified

MV* frameworks also simplify some of the tasks we as developers deal with on a regular basis. Take, for instance, repetitively printing out the data from a list, complete with HTML markup, to the screen. That's a run-of-the-mill task, but the mechanics involved take a good deal of code to pull it off. Moreover, we find ourselves repeating the same code over and over every time we need to do this.

Let's consider the employee directory from the beginning of the chapter. One of the requirements is to be able to add an employee's data as an entry to a list. If you had to code all the mechanics by hand, you'd find yourself manually creating DOM elements for each entry in a JavaScript loop. The code wouldn't be pretty, and chances are it wouldn't be reusable for other tasks.

MV* frameworks take the drudge work out of tasks like this. Take the code in listings 2.12 and 2.13. This is how we're adding entries to our list using Knockout. Listing 2.12 is from the HTML side of things. For brevity, this isn't the entire source code, only the portion to add to the directory's employee listing. The complete source code can be found in appendix A.

Listing 2.12 Employee list HTML

```
<ul class="entry-list" id="entryList"
   data-bind="foreach: entries">
</ul>
```

An empty UL and a foreach binding to iteratively build our list

Notice that we don't have any JavaScript code in our HTML. The following listing shows the ViewModel backing this section of the template.

Listing 2.13 Employee list JavaScript

```
self.entries = ko.observableArray();
self.addEntry = function(e) {
    var newEntry = {
          firstName : self.entry.firstName(),
          lastName : self.entry.lastName(),
          title : self.entry.title(),
          phone : self.entry.phone(),
          email : self.entry.email()
    };
    self.entries.push(newEntry);
};
```

Special Knockout array that keeps the array data in sync with the HTML

Create a new entry whenever the Add button is clicked

Add the entry to the array, and Knockout takes care of the rest

With a few subtle attributes and a minimal amount of code, we were able to accomplish our list of management needs. Taking the grunt work out of routine tasks like this greatly simplifies life as a programmer.

2.3.3 Productivity gains

From a development standpoint, being able to devote your time and energy to your business logic is a definite boost to productivity. When we do decide to use an external library or framework, we're removing the burdens of having to maintain that part of the code base ourselves. We also use the expertise of their authors in the areas that the particular framework covers. Sure, you could create your own routines to do the same thing, but it would take an enormous effort to get it to the level of the MV* implementations out there.

You also have an incredible amount of community-based knowledge on the web for most libraries/frameworks, should you run into problems. Most authors of MV* have mechanisms for reporting bugs too. This means that periodically code fixes are tested and released, without you having to spend your time on the issue.

If you're doing everything yourself, you're maintaining your own business logic, *plus* the extra bug fixes and testing for all of the structural code provided from external libraries/frameworks.

2.3.4 Standardization

As you'll hear me repeat throughout this book, writing a robust web application with a clean, scalable code base is already difficult. This difficulty can be compounded in a single-page app. So the last thing you need is to have everyone on the development team writing code in completely different styles.

You want to be able to read your teammate's code as if it were your own. Otherwise, you'll continuously waste time deciphering some "foreign" style of coding before you can get around to updating it. Even if you're alone, not working with a team, having uniform code standards will help you when it's time to revisit something you wrote to make changes.

MV* libraries and frameworks have certain conventions that must be followed in order to use the software. This will compel you to write your application's code in a more formal, standardized way.

2.3.5 Scalability

As discussed previously, MV* frameworks inherently promote the separation-of-concerns concept. This, in turn, also makes a project more scalable, because loosely coupled objects can be reworked with minimal effect on other objects.

Objects can also be swapped out entirely to make room for new functionality without causing a huge ripple effect throughout the project. This allows the project to grow more gracefully, because code changes tend to have much less negative impact.

Now that you've seen some of the core MV* features, which style seems easier? Which is more difficult to use? You'll have to be the judge. Some people don't like having the declarative style of bindings freely mixed with the HTML page itself as you see with MVVM. Others prefer it over having to use so much boilerplate code to create the view.

Although you'll ultimately have to decide which framework is right for you and your project, section 2.4 will give you a few things to think about as you're making your decision.

2.4 Choosing a framework

After you've decided you do want to use an MV* library or framework, you have a lot to choose from. Even if you decide on a particular style you like better, you'll still have a lot of candidates to weed through. Just to give you an idea, here are some popular client-side MV* options available at the time of this writing:

- AngularJS (https://angularjs.org)
- Agility.js (http://agilityjs.com)
- Backbone.js (http://backbonejs.org)
- CanJS (http://canjs.com)
- Choco (https://github.com/ahe/choco)
- Dojo Toolkit (http://dojotoolkit.org)
- Ember.js (http://emberjs.com)
- Ext JS (www.sencha.com/products/extjs)
- Jamal (https://github.com/adcloud/jamal)
- JavaScriptMVC (http://javascriptmvc.com)
- Kendo UI (www.telerik.com/kendo-ui)
- Knockout (http://knockoutjs.com)
- Spine (http://spinejs.com)

As you can see, you have quite a few choices. And those are the libraries/frameworks themselves. If you decide to choose a framework that's not all-inclusive, the lists of

libraries and frameworks to handle the other features, such as routing and view management, are nearly as long. In a few years, we went from relatively few choices to an overwhelming number of them.

It's rather unproductive to try to point out which ones are "better" than others. It's all a matter of opinion. Also, because they're all different and have a different number of features and styles, it's hard to make an apples-to-apples comparison. But I can offer a list of things to keep in mind as you're making your decision:

- *A la carte or one-stop shopping*—This is completely subjective, but do you want a framework that has everything built in? Or do you prefer something that's as small as possible and focused on a few core features? There are arguments for both. Some people would rather not have to worry about finding other libraries for missing features. That's just more dependencies to worry about and more potential points of failure. You'd also have to be versed in software from various providers instead of just one. But some people point out the other side of the coin: by going with an all-inclusive solution, you're "putting all your eggs in one basket." If the framework ever stops being supported, it'll be tougher to replace everything than a single supporting library. Smaller offerings, such as Knockout and Backbone.js, are great if you want to go minimal, but you'll have to look elsewhere to fill in any gaps when writing your SPA. Frameworks such as Ember.js, Kendo UI, and AngularJS plug most gaps but hide a lot of what's going on with their framework magic. This is a negative for some people who want more control.
- *Licensing and support*—Budget is always a factor. For your project, do you have money to spend on a framework, or do you need something that's free? Are you required by your company to purchase a commercial product? Does your company require you to be able to purchase a certain level of support for any software used in its projects? Is your project mission critical? Is a minimum turnaround required for bug fixes and updates?
- *Programming style preference*—Knockout and Kendo UI fall squarely in the MVVM camp. Others, including Backbone.js and Ember.js, are more MVC and/or MVP. AngularJS is a little more MVVM but still retains some MVC-like features. Any of these can be used to create large, robust applications. Your selection boils down to your personal preference after you've tried a few of them.
- *Learning curve*—This might be a minor point to some, because given enough time you can learn to use any framework. Some are definitely more difficult to wrap your head around than others. You might not have months to get up to speed.
- *Number of bugs and fix rate*—All software has a certain number of bugs at any given time. That's just the way things are. But what you can factor into your decision is the percentage of high and critical bugs the software experiences over time. Also, how fast are they being fixed? If a large number of important bugs have been sitting there for a long time, that's probably a red flag.

- *Documentation*—How good is their documentation? How up to date is it? Some MV* providers offer free online videos and interactive training. Are there code examples to go along with the API documentation?
- *Maturity*—We can't judge how good the framework is by how mature it is. We can, however, get a warm and fuzzy that it's here for the foreseeable future if it's pretty mature. If software is fairly new, it's probably still going through "growing pains." That might be tolerable for applications that aren't a high priority. If the software is constantly changing, though, it would be nearly impossible to create a mission-critical application with it.
- *Community*—This aspect is sometimes overlooked, but if you plan on including third-party software as a dependency, it's nice if it has a large community following. There's strength in numbers. Sooner or later, you'll run into situations that aren't covered in the documentation. Finding help in online forums and blogs can be a lifesaver.
- *How opinionated is it?*—For routine tasks, such as creating objects and lists or sending, receiving, and processing server requests, how flexible is it? Does it limit you by imposing strict guidelines (and are you OK with that)? How well does it play with any other libraries/frameworks in your arsenal?
- *POC (proof of concept)*—Once you've narrowed down your choices to a select few, do a POC for each to get a feel for it in practice. You'll always encounter situations in your real project that you didn't anticipate and be forced to search for workarounds. That's just the nature of the beast. But by doing a simple POC with at least basic CRUD functionality, you'll be able to make a decision. Preferably, your CRUD operations will include a list of objects so you can get a feel for how easy it is to manage a collection as well.

As you can see, you have many factors to think about when choosing an MV* framework. But you're now acquainted with the traditional design patterns and the basic core concepts. You've also been exposed to some of the design differences of MV* libraries/frameworks. With that and this list of points to consider, you'll be able to make a more informed choice when the time comes.

2.5 Chapter challenge

Now here's a challenge for you, to see what you've learned in this chapter. Let's see if you, on your own, can use one-way bindings to create a simple view. Let's pretend that your local library wants to begin offering e-books online and has reached out to the community for help. The library already has converted its first set of books to e-books but needs someone with web development skills to set up the e-book site. Pick any one of the three MV* frameworks from this chapter (or a different one if you prefer) and create a view that's contains a list of a few books. The view should have the following format:

- *Header*—The header should contain the library name, address, and phone number. It should also display the name of the user logged in. For the user,

create a JavaScript variable and use your name as its value. Then create a simple binding to display this value in the view's header.

- *Body*—Create a list of book objects (book title, author, and simple description) in JavaScript. In the body, choose a binding that prints each book in the list iteratively.

2.6 Summary

Understanding the information in this chapter will help you going forward. Let's review:

- The traditional design patterns that had a major influence on MV* libraries/frameworks are MVC (Model-View-Controller), MVP (Model-View-Presenter), and MVVM (Model-View-ViewModel).
- The model represents your data. In MVVM, this object is mainly just data. In the other two patterns, the model also contains other kinds of logic, including logic to manage the data.
- The view represents the part of the application that the user sees and interacts with.
- The third object, the controller or presenter or ViewModel, is an intermediary object of one degree or another, keeping the model and the view decoupled but interactive.
- Each pattern must be adaptable to its environment. The authors of MV* libraries/frameworks have had to take various liberties with traditional patterns in order to create solutions that work in a browser setting.
- Some basic MV* concepts to know are models, bindings, templates, and views.
- You should keep a variety of considerations in mind when choosing an MV* framework: a la carte or one-stop shopping, licensing and support, programming style preference, learning curve, bugs and fix rate, documentation, maturity, community support, and how opinionated it is.
- When you narrow your choices to two or three, try doing a proof of concept with each to get a feel for its use in your project.

The next step in the development of web-based software, single-page web applications deliver the sleekness and fluidity of a native desktop application in a browser. If you're ready to make the leap from traditional web applications to SPAs but don't know where to begin, this book will get you going.

SPA Design and Architecture teaches you the design and development skills you need to create SPAs. You'll start with an introduction to the SPA model and see how it builds on the standard approach using linked pages. The author guides you through the practical issues of building an SPA, including an overview of MV* frameworks, unit testing, routing, layout management, data access, pub/sub, and client-side task automation. This book is full of easy-to-follow examples you can apply to the library or framework of your choice.

What's inside

- Working with modular JavaScript
- Understanding MV* frameworks
- Layout management
- Client-side task automation
- Testing SPAs

This book assumes you are a web developer and know JavaScript basics.

Embracing modularity and dependency management

Modularity is another key element to building large, modern JavaScript applications. Using a modular approach leads to better-designed code that's more manageable and testable. The following chapter provides key insight into using JavaScript modules with an eye towards large-scale application design.

Chapter 5 from *JavaScript Application Design*
by Nicolas G. Bevacqua

Embracing modularity and dependency management

This chapter covers

- Working with code encapsulation
- Understanding modularity in JavaScript
- Incorporating dependency injection
- Using package management
- Trying out ECMAScript 6

Now that we're done with the Build First crash course, you'll notice a decline in Grunt tasks, though you'll definitely continue to improve your build. In contrast, you'll see more examples discussing the tradeoffs between different ways you can work with the JavaScript code underlying your apps. This chapter focuses on *modular design*, driving down the code complexity of applications by separating concerns into different modules of interconnected, small pieces of code that do one thing well and are easily testable. You'll manage complexity in asynchronous code flows, client-side JavaScript patterns and practices, and various kinds of testing in chapters 6, 7, and 8, respectively.

Part 2 boils down to increasing the quality in your application designs through separation of concerns. To improve your ability to separate concerns, I'll teach you

all about modularity, shared rendering, and asynchronous JavaScript development. To increase the resiliency of your applications, you should test your JavaScript, as well, which is the focus of chapter 8. While this is a JavaScript-focused book, it's crucial that you understand REST API design principles to improve communication across pieces of your application stack, and that's exactly the focus of chapter 9.

Figure 5.1 shows how these bits and pieces of the second half of the book relate to each other.

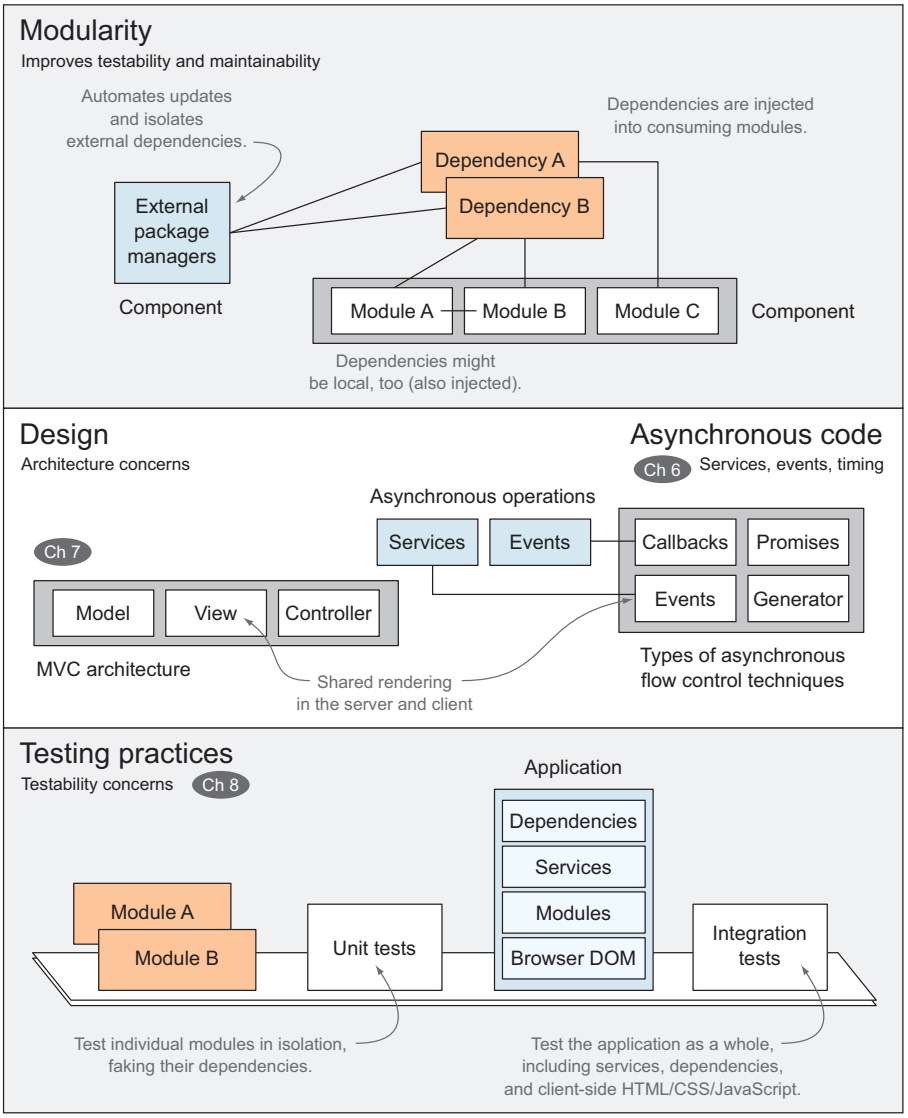


Figure 5.1 Modularity, good architecture, and testing are fundamentals of designing maintainable applications.

Applications typically depend on external libraries (such as jQuery, Underscore, or AngularJS), which should be handled and updated by using package managers, rather than manually downloaded. Similarly, your application can also be broken down into smaller pieces that interact with each other, and that's another focus of this chapter.

You'll learn the art of code encapsulation, treating your code as self-contained components; designing great interfaces and arranging them precisely; and information hiding to reveal everything the consumer needs, but nothing else. I'll spend a good number of words explaining elusive concepts such as *scoping*, which determines where variables belong; the `this` keyword, which you must understand; and *closures*, which help you hide information.

Then we'll look at dependency resolution as an alternative to maintaining a sorted list of script tags by hand. Afterward, we'll jump to package management, which is how you'll install and upgrade third-party libraries and frameworks. Last, we'll look at the upcoming ECMAScript 6 specification, which has a few nice new tricks in store for building modular applications.

5.1 Working with code encapsulation

Encapsulation means keeping functionality self-contained and hiding implementation details from consumers of a given piece of code (those who access it). Each piece, whether a function or an entire module, should have a clearly defined responsibility, hide implementation details, and expose a succinct API to satisfy its consumers' needs. Self-contained functionality is easier to understand and change than code that has many responsibilities.

5.1.1 Understanding the Single Responsibility Principle

In the Node.js community, inspired by the UNIX philosophy of keeping programs concise and self-contained, packages are well known for having a specific purpose. The high availability of coherent packages that don't go overboard with features plays a big role in making the `npm` package manager great. For the most part, package authors accomplish this by following the Single Responsibility Principle (SRP): build packages that do one thing, and do it well. SRP doesn't apply only to packages as a whole; you should follow SRP at the module and method levels, as well. SRP helps your code stay readable and maintainable by keeping it simple and concise.

Consider the following use case. You need to build a component that takes a string and returns a hyphenated representation. It will be helpful when generating semantic links in web applications such as blogging platforms. This component might take blog post titles such as `'Some Piece Of Text'`, and convert them to `'some-piece-of-text'`. This is called *slugging*.

Suppose you start with the following listing (available as ch05/01_single-responsibility-principle in the samples). It uses a two-step process in which it first normalizes all nonalphanumeric character sequences into single dashes and then removes leading and trailing dashes. Then it lowercases the string. Exactly what you need but nothing else.

Listing 5.1 Converting text using slugging

```
function getSlug (text) {
    var separator = /[^a-z0-9]+/ig;
    var drop = /^-|-$/g;
    return text
        .replace(separator, '-')
        .replace(drop, '')
        .toLowerCase();
}
var slug = getSlug('Some Piece Of Text');
// <- 'some-piece-of-text'
```

The first expression, `/[^a-z0-9]+/ig` is used to find sequences of one or more characters that aren't alphanumerical, such as spaces, dashes, or exclamation points. These expressions are replaced by dashes. The second expression looks for dashes at either end of the string. Combining these two, you can build a URL-safe version of blog post titles.

Understanding regular expressions

Although you don't need to know regular expressions to understand this example, I encourage you learn the basics. Regular expressions are used to find patterns in strings, and they can also be used to replace those occurrences with something else. These expressions are supported in virtually all major languages.

Expressions such as `/[^a-z0-9]+/ig` can be confusing to look at, but they aren't that hard to write! My blog has an entry-level article you can read if the subject interests you.[a]

[a] You can find the article on my blog at http://bevacqua.io/bf/regex.

In the previous example, the `separator` variable is a simple regular expression that will match sequences of non-letter, non-numeric characters. For example, in the `'Cats, Dogs and Zebras!'` string, it will match the first comma and space as a single occurrence, both spaces around `'and'`, and the `'!'` at the end. The second regular expression matches dashes at either end of the string, so that the resulting slug begins and ends with words, especially because you're converting any nonalphanumeric characters into dashes in the previous step. Combining these two steps is enough to produce a decent slugging function for your component.

Imagine a feature request for which you need to add a timestamp of the publication date to the slug. An optional argument in the slugging method to turn on this functionality might be tempting, but it would also be wrong: your API would become more confusing to use, harder to refactor (change its code without breaking other components, detailed in chapter 8 when we discuss testing), and even more difficult to document. It would be more sensible to build your component by following the SRP principle using a composition pattern instead. *Composition* only means applying

functions in sequence, rather than mashing their functionality together. So first you'd apply slugging and then you could add a timestamp to the slugs, as shown in the following code snippet:

```
function stamp (date) {
    return date.valueOf();
}
var article = {
  title: 'Some Piece Of Text',
  date: new Date()
};
var slug = getSlug(article.title);
var time = stamp(article.date);
var url = '/' + time + '/' + slug;
// <- '/1385757733922/some-piece-of-text'
```

Now, imagine that your Search Engine Optimization (SEO) expert comes along, and he wants you to exclude irrelevant words from your URL slugs so you get better representation in search results. You might be tempted to do that right in the `getSlug` function, but here are a few reasons why that would be wrong in this case, too:

- It would become harder to test the slugging functionality on its own, because you'd have logic that doesn't have anything to do with the slugging.
- The exclusion code might become more advanced as time goes on, but it'd still be contained in `getSlug`.

If you're cautious, you'll code a function aimed at the expert's requirements, which looks like the following code snippet:

```
function filter (text) {
    return text.replace(keywords, '');
}
var keywords = /\bsome|the|by|for|of\b/ig; // match stopwords
var filtered = filter(article.title);
var slug = getSlug(filtered);
var time = stamp(article.date);
var url = '/' + time + '/' + slug;
// <- '/1385757733922/piece-text'
```

That looks fairly clean! By giving each method a clear responsibility, you extended your functionality without complicating matters too much. In addition, you uncovered the possibility of reuse. You might use the SEO expert's filtering functionality all over an application, and that would be easy to extract from your slugging module, because it doesn't depend on that. Similarly, testing each of these three methods will be easy. For now, it should be enough to say that keeping code succinct and to the point and doing exactly what the function name implies is one of the fundamental aspects of maintainable, testable code. In chapter 8 you'll learn more about unit testing.

Splitting functionality in a modular way is important, but it's not enough. If you're building a typical component, which has a few methods but shouldn't expose its variables, you need to hide this information from the public interface. I'll discuss the importance of information hiding next.

5.1.2 Information hiding and interfaces

As you're building out an application, code will invariably grow in both volume and complexity. This can eventually turn your code base into an unapproachable tangle, but you can help it by writing more straightforward code and making it easier to follow the flow of code. One way to drive down the complexity creep is to hide away unnecessary information, keeping it inaccessible on the interface. This way only what matters gets exposed; the rest is considered to be irrelevant to the consumer, and it's often referred to as *implementation details*. You don't want to expose elements such as state variables you use while computing a result or the seed for a random number generator. This has to be done at every level; each function in every module should attempt to hide everything that isn't relevant to its consumers. In doing this, you'll do fellow developers and your future self a favor by reducing the amount of guesswork involved in figuring out how a particular method or module works.

As an example, consider the following listing illustrating how you might build an object to calculate a simple average sum. The listing (found as ch05/02_information-hiding in the samples) uses a constructor function and augments the prototype so `Average` objects have an `add` method and a `calc` method.

Listing 5.2 Calculating an average sum

```
function Average () {
    this.sum = 0;
    this.count = 0;
}

Average.prototype.add = function (value) {
    this.sum += value;
    this.count++;
};

Average.prototype.calc = function () {
    return this.sum / this.count;
};
```

All that's left to do is create an `Average` object, add values to it, and calculate the average. The problem in this approach is that you might not want people directly accessing your private data, such as `Average.count`. Maybe you'd rather hide those values from the API consumers using the techniques we'll cover soon. An even simpler approach might be to ditch the object entirely and use a function instead. You could use the `.reduce` method (found on the Array prototype, new in ES5) to apply an accumulator function on an array of values to calculate the average:

```
function average (values) {
    var sum = values.reduce(function (accumulator, value) {
        return accumulator + value;
    }, 0);

    return sum / values.length;
}
```

The upside of this function is that it does exactly what you want. It takes an array of values, and it returns the average, as its name indicates. In addition, it doesn't keep any state variables the way your prototypical implementation did, effectively hiding any information about its inner workings. This is what's called a *pure function*: the result can only depend on the arguments passed to it, and it can't depend on state variables, services, or objects that aren't part of the argument body. Pure functions have another property: they don't produce any side effects other than the result they provide. These two properties combined make pure functions good interfaces; they are self-contained and easily testable. Because they have no side effects or external dependencies, you can refactor their contents as long as the relationship between input and output doesn't change.

FUNCTIONAL FACTORIES

An alternative implementation might use a *functional factory*. That's a function that, when executed, returns a function that does what you want. As you'll better understand in the next section, anything you declare in the factory function is private to the factory, and the function that resides within. This is easier to understand after reading the following code:

```
function averageFactory () {
    var sum = 0;
    var count = 0;
    return function (value) {
        sum += value;
        count++;
        return sum / count;
    };
}
```

The `sum` and `count` variables are only available to instances of the function returned by `averageFactory`; furthermore, each instance has access only to its own context, those variables that were declared within that instance, but not to the context of other instances. Think of it like a cookie cutter. The `averageFactory` is the cookie cutter, and it cuts cookies (your function) that take a value and return the cumulative average (so far). As an example, here's how its use might look:

```
var avg = averageFactory();
// <- function
avg(1);
// <- 1
avg(3);
// <- 2
```

Much like using your cookie cutter to cut out new cookies won't affect existing cookies, creating more instances won't have any effect on existing ones. This coding style is similar to what you did previously using a prototype, with the difference that `sum` and `count` can't be accessed from anywhere other than the implementation. Consumers can't access these variables, effectively making them an implementation detail of the API. Implementation details don't only introduce noise; they can also potentially

present security concerns: you wouldn't want to grant the outside world the ability to modify the inner state of your components.

Understanding *variable scopes*, which define where variables are accessible, and `this` keyword, which provides context about the caller of a function, is essential in building solid structures that can hide information properly. Properly scoping variables enables you to hide the information that consumers of an interface aren't supposed to know about.

5.1.3 Scoping and this keyword

In his undisputed classic, *JavaScript: The Good Parts* (O'Reilly Media, 2008),[1] Douglas Crockford explains many of the quirks of the language, and encourages us to avoid the "bad parts," such as `with` blocks, `eval` statements, and type-coercing equality operators (`==` and `!=`). If you've never read his book, I recommend you do so sooner rather than later. Crockford says that `new` and `this` are tricky to understand, and he suggests avoiding them entirely. I say you need to understand them. I'll describe what `this` represents, and then I'll explain how it can be manipulated and assigned. In any given piece of JavaScript code, the context is made up of the current function scope, and `this`.

If you're used to server-side languages, such as Java or C#, then you're used to thinking of a scope: the bag where variables are contained, which starts and ends whenever a curly brace is opened and closed, respectively. In JavaScript, scoping happens at the function level (called *lexical scoping*), rather than at the block level.

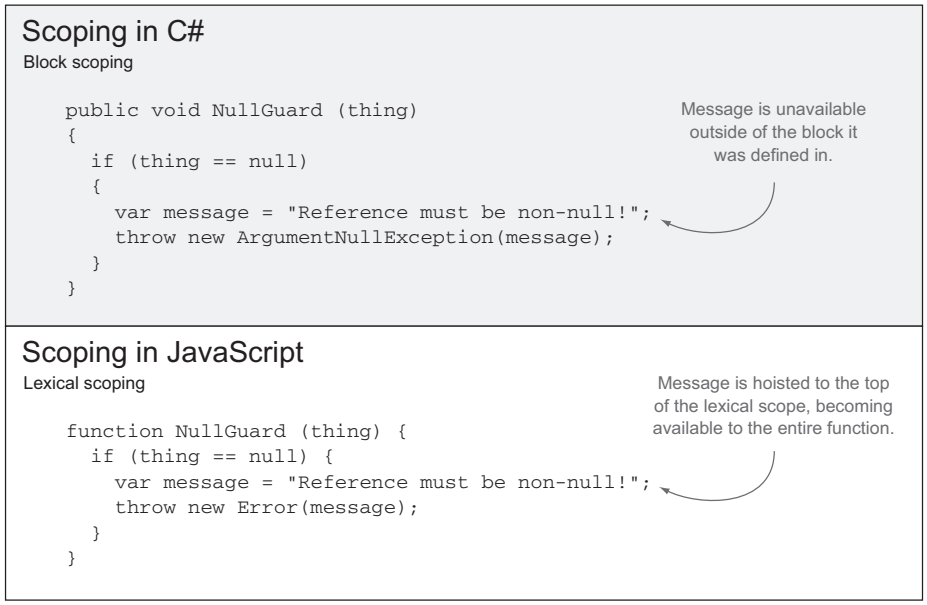


Figure 5.2 Discrepancies in scoping across languages

[1] You can find *JavaScript: The Good Parts* at Amazon following this link: http://bevacqua.io/bf/goodparts.

Figure 5.2 disambiguates lexical scoping from block scoping by comparing C#, which has block scoping (other examples include Java, Perl, C, and C++) with JavaScript, which has lexical scoping (R is another example).

In the figure, a `message` variable is used in both examples. In the first example, `message` is only available inside the `if` statement block, while in the second example `message` is available to the entire function, thanks to lexical scoping. As you'll learn, this has both benefits and drawbacks.

VARIABLE SCOPING IN JAVASCRIPT

An understanding of how scopes work will set you up to understand the module pattern, which we'll visit in section 5.2 as a way of componentizing your code base. In JavaScript, `function` is a first-class citizen, and it's treated like any other object. Nested functions each come with their own scope, and inner functions have access to the parent scope up until the global space. Consider the `getCounter` function in the following code:

```
function getCounter () {
    var counter = 0;
    return function () {
        return counter++;
    };
}
```

In this example, the `counter` variable is context-bound to the `getCounter` function. The returned function can access `counter`, because it's part of the parent scope. But nothing outside `getCounter` can create a reference to `counter`; access to it has been shut down and only the privileged children of `getCounter` can manipulate it. If you introduce a `console.log(this)` statement at either scoping level, you'll see in both cases the global `Window` object instance is referenced. This is the true "bad part;" by default, the `this` keyword will be a reference to the global object, as demonstrated in the following listing.

Listing 5.3 Understanding the `this` keyword

```
function scoping () {
    console.log(this);

    return function () {
        console.log(this);
    };
}
scoping()();
// <- Window
// <- Window
```

There are different ways we can manipulate the `this` keyword. The most common way to assign a `this` context is to invoke methods on an object. For example, when doing `'Hello'.toLowerCase()`, `'Hello'` will be used as the `this` context for the function call.

GETTING TO THE CALL SITE

When functions are invoked directly as properties on an object, the object will become the `this` reference. If the method is in the object's prototype—for example `Object.prototype.toString`—`this` will also be the object the method has been invoked on. Note that this is a fragile behavior; if you get a direct reference to a method and invoke that, then `this` won't be the `parent` anymore but rather the global object once again. To illustrate, let me show you another listing.

Listing 5.4 Scoping the `this` keyword

```
var parent = {
    method: function () {
        console.log(this);
    }
};
parent.method();
// <- parent
var parentless = parent.method;
parentless();
// <- Window
```

When the method's call site is on a parent object, then that object is used.

If there's no parent object, then we fall back to the default context.

Under strict mode, `this` will default to `undefined`, instead of `Window`. Outside strict mode, `this` is always an object; it's the provided object if it's called with an object reference; it's a boxed representation if it's called with a primitive boolean, string, or numeric value; or it's the global object (again, `undefined` under strict mode) if it's called with either `undefined` or `null`, either by getting a direct reference to the method or by using any one of these: `.apply`, `.call`, or `.bind`. The value passed as `this` to a function in strict mode isn't boxed into an object. We'll get to what else strict mode does shortly.

Other than what happens out of the box when invoking functions, you can use different methods to assign a value to `this`; it's not entirely out of your control. In fact, you could use `.bind` to create a function that will always have the `this` value provided to it. Alternative ways of executing a method include `.apply`, `.call`, and the `new` operator. Here's a cheat sheet so you can see the methods in action:

```
Array.prototype.slice.call([9, 5, 7], 1, 2)
// <- [5]

String.prototype.split.apply('13.12.02', ['.'])// <- ['13', '12', '02']

var data = [1, 2];
var add = Array.prototype.push.bind(data, 3);

add(); // effectively the same as data.push(3)
add(4); // effectively the same as data.push(3, 4)

console.log(data);
// <- [1, 2, 3, 3, 4]
```

In JavaScript, variables fill a scope in the following order:

- Scope context variables: this and arguments
- Named function parameters: function (these, variable, names)
- Function expressions: function something () {}
- Local scope variables: var foo

If you're not experimenting or following along with a JavaScript interpreter by your side, make sure to look at the code sample (ch05/03_context-scoping); I've included these examples in the source code provided with the book, and they have a few inline comments if you have trouble understanding. Let's now discuss what the strict mode entails.

5.1.4 Strict mode

When enabled, strict mode modifies semantics in the way your code works, reducing the leniency toward missing `var` statements and similarly error-prone practices, sort of complementary to using a linter.[2] Strict mode can be enabled on individual functions or on an entire script.

For client-side code, the function form is preferred. To turn on strict mode, put the `'use strict';` statement (double quotes work, too) at the top of a file or function:

```
function () {
    'use strict';
    // here lies strict mode
}
```

Aside from `this` defaulting to `undefined`, rather than the global object, strict is less tolerant of mistakes, turning them into errors rather than correcting them. Restrictions also include banning the `with` statement, octal notation, and preventing keywords such as `eval` and `arguments` to be assigned.

```
'use strict';
foo = 'bar' // ReferenceError foo is not defined
```

Under strict mode, the engine also throws an exception if you attempt to write on read-only properties, delete undeletable properties, instantiate an object with duplicate property keys, or declare a function with duplicate argument names. This kind of intolerance helps catch issues due to sloppy coding.

The last quirk I want to cover while we're on the topic of scoping is something that's commonly referred to as hoisting. Understanding hoisting is important if you're to write complex JavaScript applications sensibly.

[2] Get a detailed explanation of strict mode in Mozilla Developer Network at http://bevacqua.io/bf/strict.

5.1.5 Variable hoisting

A large number of JavaScript interview questions can be answered with an understanding of scoping, how `this` works, and hoisting. We've covered the first two, but what exactly is hoisting? In JavaScript, *hoisting* means that variable declarations are pulled to the beginning of a scope. This explains the unexpected behavior you can observe in certain situations.

Function expressions are hoisted entirely: the function body is also hoisted, not only their declaration. If I had a single thing to take away from *The Good Parts*, it would be learning about hoisting; it changed the way I write code, and reason about it.

Hoisting is the reason invoking function expressions before declaring them works as expected. Assigning functions to a variable won't do the trick, because the variable won't be assigned by the time you want to invoke the function. The following code is one example; you'll find more examples in the accompanying source code, listed as ch05/04_hoisting:

```
var value = 2;

test();

function test () {
  console.log(typeof value);
  console.log(value);
  var value = 3;
}
```

You might expect the method to print `'number'` first, and `2` afterward, or maybe `3`. Try running it! Why does it print `'undefined'` and then `undefined`? Well, hello hoisting! It'll be easier to picture if you rearrange the code the way it ends up after hoisting takes place. Let's look at the following listing.

Listing 5.5 Using hoisting

```
var value;

function test () {
  var value;
  console.log(typeof value);
  console.log(value);
  value = 3;
}

value = 2;
test();
```

The `value` declaration at the end of the `test` function got hoisted to the top of the scope, and it's also why `test` didn't give a `TypeError` exception, warning that `undefined` isn't a function. Keep in mind that if you used the variable form of declaring the `test` function, you would, in fact, have gotten that error, because although `var test` would be hoisted, the assignment wouldn't be, effectively becoming the code in the following listing.

Listing 5.6 Hoisting `var test`

```
var value;
var test;

value = 2;
test();

test = function () {
  var value;
  console.log(typeof value);
  console.log(value);
  value = 3;
};
```

The code in listing 5.6 won't work as expected, because `test` won't be defined by the time you want to invoke it. It's important to learn what gets hoisted and what doesn't. If you make a habit of writing code as if it were already hoisted, pulling variable declarations and functions to the top of their scope, you'll run into fewer problems than you might run into otherwise. At this point you should feel comfortable with scoping and the `this` keyword. It's time to talk about closures and modular patterns in JavaScript.

5.2 JavaScript modules

Up to this point, you've looked at the single responsibility principle, information hiding, and how to apply those in JavaScript. You also have a decent idea of how variables are scoped and hoisted. Let's move on to closures. These will help you create new scopes and prevent variables from leaking information.

5.2.1 Closures and the module pattern

Functions are also referred to as closures, particularly when focusing on the fact that functions create new scopes. An IIFE is a function that you execute immediately. The term IIFE stands for Immediately-Invoked Function Expression. Using an IIFE is useful when all you want is a closure. The following code is an example IIFE:

```
(function () {
    // a new scope
})();
```

Note the parentheses wrapping the function. These tell the interpreter you're not only declaring an anonymous function, but also using it as a value. These expressions can also be used in assignments, which are useful if you need variables accessible by the exported return value. This is commonly referred to as the module pattern, as shown in the following code (labeled ch05/05_closures in the samples):

```
var api = (function () {
    var local = 0; // private and in-place!
    var publicInterface = {
        counter: function () {
            return ++local;
        }
```

```
    };
    return publicInterface;
})();
api.counter();
// <- 1
```

A common variant to the previous code doesn't rely on anything outside of the closure, but instead imports the variables it's going to use. If it wants to expose a public API, then it imports the global object. I tend to favor this approach because everything is nicely wrapped by a closure, and you can instruct JSHint to blow up on issues due to undeclared variables. Without a closure and JSHint, these would inadvertently become globals. To illustrate, look at the following code:

```
(function (window) {
    var privateThing;

    function privateMethod () {
    }

    window.api = {
        // public interface
    };
})(window);
```

Let's consider *prototypal modularity*, which augments a prototype rather than using closures, as a complementary alternative to IIFE expressions. Using prototypes provides performance gains, as many objects can share the same prototype and adding functions on the prototype provides the functionality to all the objects that inherit from it.

5.2.2 Prototypal modularity

Depending on your use case, prototypes might be exactly what you need. Think of prototypes as JavaScript's way of declaring classes, even though it's an entirely different model, because prototypes are simply links, and you can't override properties unless you replace them entirely (and do the overriding by hand). In short, don't try to treat prototypes as classes, because it will assuredly result in maintainability issues. Prototypes are most useful when you expect to have multiple instances of your module. For example, all JavaScript strings share the `String` prototype. A good use for prototypes is when interacting with DOM nodes. Sometimes I find myself declaring prototypal modules inside a closure and then keeping private state in the closure, outside the prototype. The following listing shows pseudo-code, but please look at the accompanying code sample listed as ch05/06_prototypal-modularity for a fully working example and to get a better understanding of the pattern.

Listing 5.7 Using pseudo-code for prototypes

```
var lastId = 0;
var data = {};

function Lib () {
    this.id = ++lastId;
    data[this.id] = {
```

```
        thing: 'secret'
    };
}

Lib.prototype.getPrivateThing = function () {
    return data[this.id].thing;
};
```

This is one way to keep data safe from consumers; many scenarios exist when data privatization isn't necessary and where allowing consumers to manipulate your instance data might be a good thing. You should wrap all of this in a closure so your private data doesn't leak out. I believe prototypes in JavaScript are most useful when dealing with DOM interaction, as we'll investigate in chapter 7. That's because when dealing with DOM objects, you usually have to work with many elements at the same time; prototypes improve performance because their methods aren't replicated on each instance, saving resources.

Now that you have a clearer understanding of how scoping, hoisting, and closures work, we can move on to how modules are meant to interact with one another. First, let's look at CommonJS modules: a way to keep code well-organized and deal with dependency injection (DI) at once.

5.2.3 CommonJS modules

CommonJS (CJS) is a specification adopted by Node.js, among others, which allows you to write modular JavaScript files. Each module is defined by a single file, and if you assign a value to `module.exports`, it becomes that module's public interface. To consume a module, you call `require` with the relative path from the consumer to the dependency.

Let's look at a quick example, labeled ch05/07_commonjs-modules in the samples:

```
// file at './lib/simple.js'
module.exports = 'this is a really simple module';

// file at './app.js'
var simple = require('./lib/simple.js');

console.log(simple);
// <- 'this is a really simple module'
```

One of the most useful advantages of these modules is that variables don't leak to the global object: you have no need to wrap your code in a closure. The variables that are declared on the top-most scope (such as the `simple` variable in the previous snippet) are merely available in that module. If you want to expose something, you need to make that intent explicit by adding it to `module.exports`.

At this point you might think I went off the trail with CJS, given that it's not supported natively in browsers any more than are CoffeeScript and TypeScript. You'll soon learn how to compile these modules using Browserify, a popular library designed to compile CJS modules to something browsers can deal with. CJS has the following benefits over the way browsers behave:

- No global variables, less cognitive load
- Straightforward process to expose an API and consume a module

- Easier to test modules by mocking dependencies
- Access to packages on npm, thanks to Browserify
- Modularity, which translates into testability
- Easy to share code between client and server, if you're using Node.js

You'll learn more about package management solutions (npm, Bower, and Component) in section 5.4. Before we get there, we'll look at *dependency management*, or how to deal with the components needed by your application, and how different libraries can help manage them.

5.3 Using dependency management

We'll discuss two kinds of dependency management here: internal and external. When talking about internal dependencies, I'm referring to those that are part of the program you're writing. Most frequently, these are a one-to-one mapping to physical files, but you might also have multiple modules in a single file. By modules I mean pieces of code that have a single responsibility, regardless of them being services, factories, models, controllers, or something else. External dependencies are, in contrast, those in which the code isn't governed by your application itself. You may own or have authored the package, but the code belongs to a different repository altogether, regardless.

I'll explain what dependency graphs are, and then we'll investigate ways of working through them, such as the caveats with resorting to the RequireJS module loader, the innocent straightforwardness made available by CommonJS, and the elegant way AngularJS (a Model-View-Controller framework built by Google) resolves dependencies while keeping everything modular and testable.

5.3.1 Dependency graphs

When writing out a module which depends on something else, the most common approach is to have your module create an instance of the object you depend on. To illustrate the point, bear with me through a little Java code; it should be easy to wrap your head around. The following listing displays a `UserService` class, which has the purpose of serving any data requests from a domain logic layer. It could consume any `IUserRepository` implementation which is tasked with retrieving the data from a repository such as a MySQL database or a Redis store. This listing is labeled ch05/08_dependency-graphs in the samples.

Listing 5.8 Using a module to create an object

```
public class UserService {
    private IUserRepository _userRepository;

    public UserService () {
        _userRepository = new UserMySqlRepository();
    }

    public User getUserById (int id) {
        return _userRepository.getById(id);
    }
}
```

But that doesn't cut it; if your service is supposed to use any repository that conforms to the interface, why are you hard-coding `UserMySqlRepository` that way? Hard-coded dependencies make it more difficult to test a module, because you wouldn't merely test against the interface, but rather against a concrete implementation. A better approach, which is coincidentally more testable, might be passing that dependency through the constructor, as shown in the following listing. This pattern is often referred to as dependency injection, which is a smart-sounding alternative to giving an object its instance variables.

Listing 5.9 Using dependency injection

```
public class UserService {
    private IUserRepository _userRepository;

    public UserService (IUserRepository userRepository) {
        if (userRepository == null) {
            throw new IllegalArgumentException();
        }
        _userRepository = userRepository;
    }

    public User getUserById (int id) {
        return _userRepository.getById(id);
    }
}
```

This way, you can build out your service the way it was intended, as a consumer of any repository conforming to the `IUserRepository` interface without any knowledge of implementation specifics. Creating a `UserService` might not sound like such a great deal, but it gets harder as soon as you take into consideration its dependencies, and its dependencies' dependencies. This is called a *dependency tree*. The following snippet is certainly unappealing:

```
String connectionString = "SOME_CONNECTION_STRING";
SqlConnectionString connString = new SqlConnectionString(connectionString);
SqlDbConnection conn = new SqlDbConnection(connString);
IUserRepository repo = new UserMySqlRepository(conn);
UserService service = new UserService(repo);
```

The code shows *inversion of control* (IoC),[3] which is a wordy definition for something rather simple. IoC means that instead of making an object responsible for the instantiation of its dependencies, or getting references to them, the object is given the dependencies through its constructor or through public properties. Figure 5.3 examines the benefits of using an IoC pattern.

[3] Read a primer on inversion of control and dependency injection by Martine Fowler at http://bevacqua.io/bf/ioc.

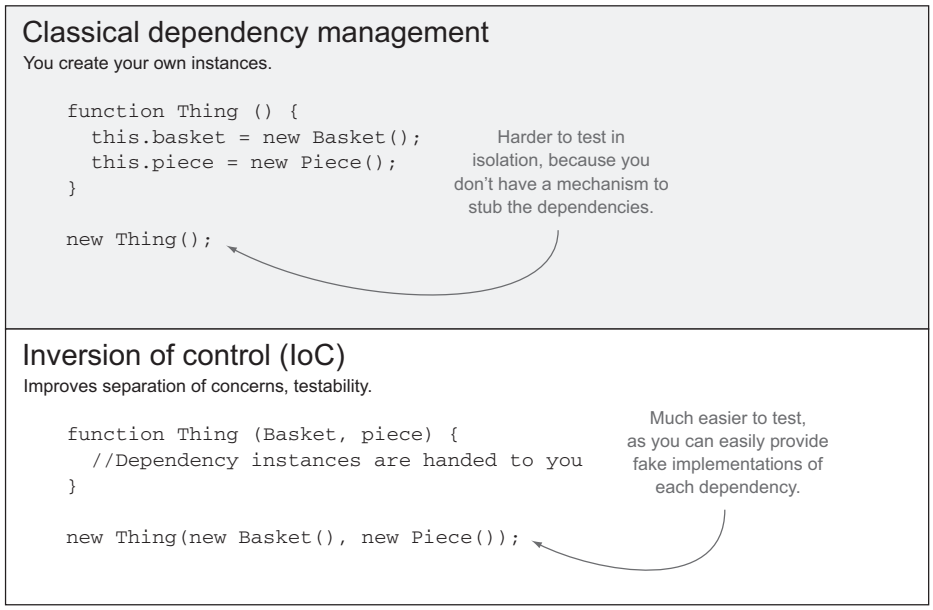


Figure 5.3 Classical dependencies compared with using IoC to improve testability

The IOC code (at the bottom of the figure) is easier to test, more loosely coupled, and easier to maintain as a result, than the classic dependency management code shown at the top of the figure.

IoC frameworks are used to address dependency resolution and mitigate dependency hell. The basic gist of these frameworks is that you ditch the `new` keyword and rely on an IoC container. The *IoC container* is a registry that has knowledge about how to instantiate your services, repositories, and any other modules. Learning how to configure a traditional IoC container (such as Spring in the case of Java, or Castle Windsor for C#) is outside of the scope of this book, but a top-level view of the issue is required to pave the road ahead.

IS IOC IMPORTANT FOR TESTABILITY?

Ultimately, the importance of avoiding hard-coded dependencies lies in the ability to easily mock them when unit testing, as you'll see in chapter 8.

Unit testing is about asserting whether interfaces work as expected, regardless of how they're implemented. *Mocks* are stubs that implement the interface, but don't do anything other than the bare minimum to conform to them. For example, a mocked user repository might always return the same hard-coded `User` object. This is useful in the context of unit testing, where you might want to test the `UserService` class on its own, but don't need details about its inner workings, much less how its dependencies are implemented!

Great! Enough Java for now, though. What does any of this have to do with JavaScript Application Design? Understanding testability principles is required if you hope to

write testable code. Although you may not agree with the Test-Driven Development movement, it's undeniable that code that isn't written with testability in mind is much harder to write tests for. When speaking about client-side JavaScript, you have an additional layer of complexity: networking. Modules aren't immediately available unless your code is bundled together the way you learned to do it in chapter 2.

Next, I'll introduce you to RequireJS, an asynchronous module loader, which is a better option than the classical approach of having an unmanaged dependency soup.

5.3.2 Introducing RequireJS

RequireJS is a JavaScript asynchronous module loader (AMD) that allows you to define modules and have them depend on one another. The following code (found as ch05/09_requirejs-usage in the samples) is an example usage of AMD, depicting a module that depends on something else:

```
require(['lib/text'], function(text) {
    var result = text('foo bar');
    console.log(result);
    // <- 'FOO BAR'
});
```

By convention, `'lib/text'` looks for the file that can be found at the `./lib/text.js` path, relative to the JavaScript directory root. That resource will be requested, interpreted, and once all dependencies have been loaded, the module's function will be invoked, getting its dependencies as arguments to the module's function, much like the Java code I talked about in section 5.3.1. The sample `'lib/text'` module is defined as follows:

```
define([], function () {
    return function (input) {
        return input.toUpperCase();
    };
});
```

Next, let's analyze where RequireJS is better than the alternatives, and where it falls short.

BENEFITS AND DRAWBACKS OF REQUIREJS

In this case, the definition uses an empty array because it has no dependencies. The returned function is the public interface provided by the `'lib/text'` module. The use of RequireJS has a few benefits:

- Dependency graph is automatically resolved. No more worrying about ordering script tags!
- Asynchronous module loading is included.
- A compile step isn't required during development.
- It's unit testable, so you only load the module that needs to be tested.
- Closures are enforced, because your module is defined in a function.

These are all true and nice to have, but drawbacks exist. If a package your code depends on isn't wrapped in AMD magic, you have no option other than adding a compile step to bundle everything together. Unless you bundle your modules together, RequireJS will create an HTTP request cascade to fetch each dependency, which would be too slow in production systems. Many of the benefits of AMD came from the lack of a compile step, so you're left with a glorified dependency graph resolver packed with the following drawbacks:

- Asynchronous loading functionality is unavailable if you use the bundler.
- It requires vendors to conform to the AMD model.
- It clutters your code with AMD wrappers.
- Production needs compilation.
- Code in release environments diverges from local development.

It's been a while since we spoke of Grunt in chapter 4, and you wouldn't want to release a bunch of unoptimized scripts! Grunt will help compile AMD modules during your builds so they don't need to be fetched asynchronously.

To compile[4] AMD modules through `r.js`, the RequireJS optimizer, using Grunt, you can use the `grunt-contrib-requirejs` package. That package allows you to pass options through to `r.js`. The following listing is the pertinent task configuration. You'll set default options that apply to every target in Grunt and tweak the `debug` target. This is useful when you'd otherwise have to repeat parts of the configuration, breaking the DRY principle.

Listing 5.10 Using Grunt to configure a module

```
requirejs: {
  options: {
    name: 'app',
    baseUrl: 'js/amd',
    out: 'build/js/app.min.js'
  },
  debug: {
    options: {
      preserveLicenseComments: false,
      generateSourceMaps: true,
      optimize: 'none'
    }
  },
  release: {}
}
```

In the debug distribution you generate a source map,[5] which helps browsers map what they're executing to the source code you used to compile it. This is useful when

[4] Check out the accompanying code sample that shows how to compile RJS modules at http://bevacqua.io/bf/requirejs.

[5] For more information on source maps, refer to this introductory article on HTML5Rocks at http://bevacqua.io/bf/sourcemap.

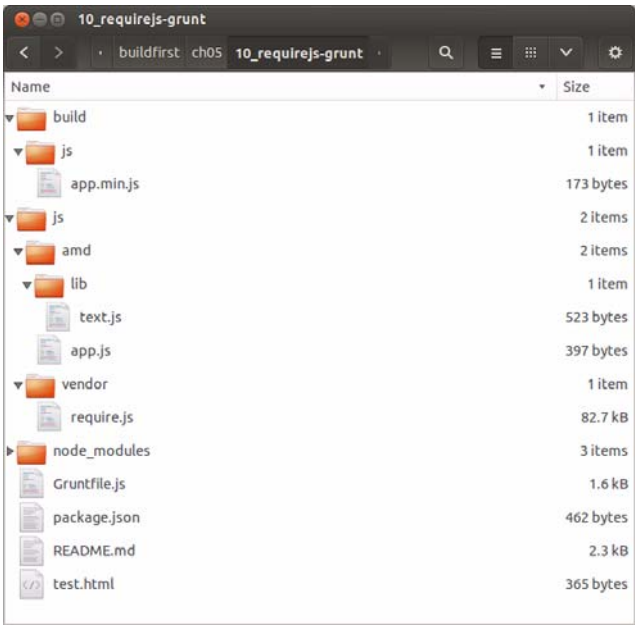


Figure 5.4 Typical file structure when using RequireJS during Grunt builds

debugging, as you'll get stack traces that point to the source code rather than hard-to-debug compilation results. The `release` target doesn't have any additional configuration, because it merely uses the defaults provided previously. It'll be easier for you to visualize the configuration if you take a look at the directory structure in the accompanying samples, which looks like the one in figure 5.4.

> **NOTE** A sample that integrates RequireJS with Grunt can be found in the book's source code at ch05/10_requirejs-grunt. It contains detailed information about the meaning of each option used to configure the RequireJS build task.

Not having to add script tags in a specific order is a nice feature to have, and you have a few ways to accomplish that. If you're not entirely sold on the AMD solution, or if you're curious, read on for an explanation of how you could bring CommonJS modules to the browser, as an alternative.

5.3.3 Browserify: CJS in the browser

In section 5.2.3 I explained the benefits of CJS, the module system used in Node.js packages. These modules also have a place in the browser, thanks to Browserify. This option is frequently pitched as an alternative to AMD, although opinions vary. As you're following a Build First approach, compiling CJS modules for the browser won't be a big deal; it's another step in your build process!

In addition to the advantages described in section 5.2.3, such as no implicit globals, CJS offers a terse alternative to AMD in that you don't need all the clutter and

boilerplate needed by AMD to define a module. A continuously improving trait in favor of CJS modules is immediate access to any package in the npm registry out of the box. In 2013, the npm registry grew by an order of magnitude (or 10x), and at the time of this writing, it boasts well more than 100,000 registered packages.

Browserify will recursively analyze all the `require()` calls in your app to build a bundle that you can serve up to the browser in a single `<script>` tag. As you might expect, Grunt has numerous plugins eager to compile your CJS modules into a Browserify bundle, and one such plugin is `grunt-browserify`. Configuring it is more akin to what you saw in chapter 2, where you provided a filename declaring the entry point of your CJS module and an output filename as well:

```
browserify: {
  debug: {
    files: { 'build/js/app.js': 'js/app.js' },
    options: { debug: true }
  },
  release: {
    files: { 'build/js/app.js': 'js/app.js' }
  }
}
```

I think most of the mental load in taking this approach won't come from Browserify, but rather learning about `require` and modularity in CJS modules. Luckily, you already used CJS modules when configuring Grunt tasks throughout part 1, and that should give you insight into CJS, as well as a bunch of code samples to look at! A fully working example of how to compile CJS modules, using `grunt-browserify`, can be found at ch05/11_browserify-cjs in the accompanying code samples. Next up, we'll look at how AngularJS deals with dependency resolution, as a third (and last) way to deal with dependency management.

5.3.4 The Angular way

Angular is an innovative client-side Model-View-Controller (MVC) framework developed at Google. In chapter 7 you'll use another popular JavaScript MVC framework called Backbone. But Angular's dependency resolver deserved a mention in this section.[6]

LEVERAGING DEPENDENCY INJECTION IN ANGULAR

Angular has a fairly elaborate dependency injection solution in place, so we won't get into the details. Luckily for us, it's abstracted well enough that it's easy to use. I've personally used many different DI frameworks, and Angular makes it feel natural: you don't even realize you're doing DI, similarly to Java and RequireJS. Let's walk together through a contrived example, which can be found at ch05/12_angularjs-dependencies in the samples. It's convenient to keep the module declaration in its own file, something like this:

```
angular.module('buildfirst', []);
```

[6] Angular's documentation has an extensive guide explaining how DI works in Angular at http://bevacqua.io/bf/angular-di.

Then each of the different pieces of a module, such as services or controllers, are registered as extensions to that module, which you previously declared. Note that you're passing an empty array to the `angular.module` function so your module doesn't depend on any other modules:

```
var app = angular.module('buildfirst');

app.factory('textService', [
  function () {
    return function (input) {
      return input.toUpperCase();
    };
  }
]);
```

Registering controllers is also similar; in the following example you'll use the `textService` service you created. This works in a similar way to RequireJS, because you need to use the name you gave to the service:

```
var app = angular.module('buildfirst');
app.controller('testController', [
  'textService',
  function (text) {
    var result = text('foo bar');
    console.log(result);
    // <- 'FOO BAR'
  }
]);
```

Next up, let's compare Angular to RJS in a nutshell.

COMPARING ANGULAR AND REQUIREJS

Angular is different from RequireJS in that, rather than acting as a module loader, Angular worries about the dependency graph. You need to add a script tag for each file you're using, unlike with AMD, which dealt with that for you.

In the case of Angular you see an interesting behavior where script order isn't all that relevant. As long as you have Angular on top and then the script that declares your module, the rest of the scripts can be in whatever order you want, and Angular will deal with that for you. You need code such as the following on top of your script tag list, which is why the module declaration needs its own file:

```
<script src='js/vendor/angular.js'></script>
<script src='js/app.js'></script>
```

The rest of the scripts, which are part of the `app` module (or whatever name you give it), can be loaded in any order, as long as they come after the module declaration:

```
<!--
    These could actually be in any order!
-->
<script src='js/app/testController.js'></script>
<script src='js/app/textService.js'></script>
```

Let's draw a few quick conclusions on the current state of module systems in JavaScript.

BUNDLING ANGULAR COMPONENTS USING GRUNT

As a side note, when preparing a build, you can explicitly add Angular and the module to the top, and then glob for the rest of the pieces of the puzzle. Here's how you might configure the `files` array passed to a bundling task, such as the ones in the `grunt-contrib-concat` or `grunt-contrib-uglify` packages:

```
files: [
    'src/public/js/vendor/angular.js',
    'src/public/js/app.js',
    'src/public/js/app/**/*.js'
]
```

You might not want to commit to the full-featured framework that is AngularJS, and you're not about to include it in your project for its dependency resolution capabilities! As a closing thought, I'd like to add that there's no right choice, which is why I presented these three methods:

- RequireJS modules, using AMD definitions
- CommonJS modules, and then compiling them with Browserify
- AngularJS, where modules will resolve the dependency graph for you

If your project uses Angular, that's good enough that you wouldn't need either AMD or CJS, because Angular provides a sufficiently modular structure. If you're not using Angular, then I'd probably go for CommonJS, mostly because of the abundance of npm packages you can potentially take advantage of.

The next section sheds light on other package managers, and as you did for npm, teaches you how to leverage them in your client-side projects.

5.4 Understanding package management

One of the drawbacks of using package managers is that they tend to organize dependencies using a certain structure. For example, npm uses `node_modules` to store installed packages, and Bower uses `bower_components`. One of the great advantages to Build First is that's not a problem, because you can add references to those files in your builds and that's that! The original location of the packages won't matter at all. That's a huge reason to use a Build First approach.

I want to discuss two popular front-end package managers in this section: Bower and Component. We'll consider the tradeoffs in each and compare them to npm.

5.4.1 Introducing Bower

Although npm is an extraordinary package manager, it isn't fit for all package management needs: virtually all of the packages published to it are CJS modules, because it's ingrained into the Node ecosystem. Although I chose to use Browserify so that I could write modular front-end code under the CJS format, this might not be the choice for every project you work on.

Bower is a package manager for the web, created at Twitter, and it's *content agnostic*, meaning it doesn't matter whether authors pack up images, style sheets, or JavaScript

code. By now you should be accustomed to the way npm tracks packages and version numbers, using the `package.json` manifest. Bower has a `bower.json` manifest that's similar to `package.json`. Bower is installed through `npm`:

```
npm install -g bower
```

Installing packages with `bower` is fast and straightforward; all you need to do is specify the name or a git remote endpoint. The first thing you'll need to do on a given project is run `bower init`. Bower will ask you a few questions (you can press Enter because the defaults are fine), and then it'll create a `bower.json` manifest for you, as in figure 5.5.

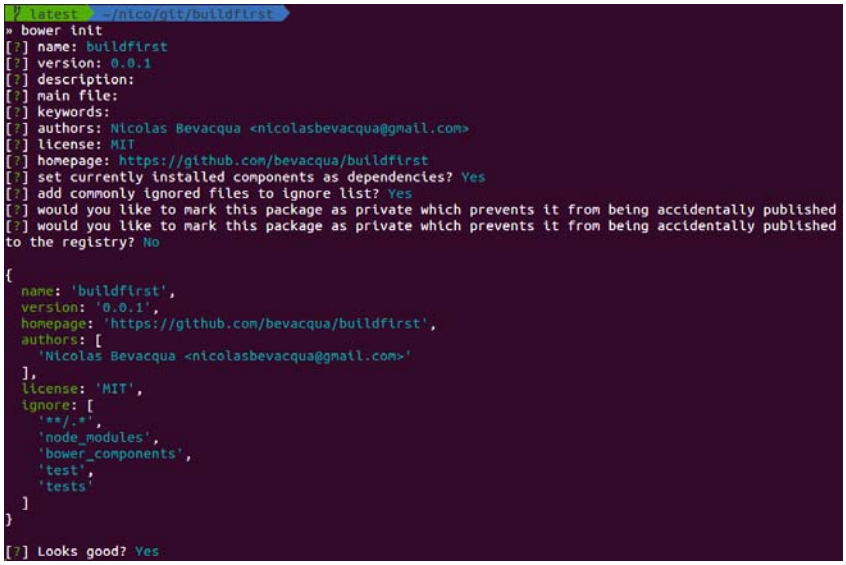


Figure 5.5 Using `bower init` to create a `bower.json` manifest file

Once that's out of the way, installing packages is a breeze. The following example installs Lo-Dash, a utility library similar to Underscore, but more actively maintained. It will download the scripts and place them in a `bower_components` directory, as shown in figure 5.6.

```
bower install --save lodash
```

Figure 5.6 Using bower `install --save` to fetch a dependency and add it to the manifest

That's it! You should have scripts in the `bower_components/lodash` directory. Including them in your builds is a matter of adding the file to your distribution configuration. As usual, this example can be found in the accompanying source code; look for `ch05/13_bower-packages`.

Bower is arguably the second-largest package manager, with close to 20,000 packages in its registry, and behind npm, which has more than 100,000. Component, another package management solution, lags behind with nearly 3,000 packages under its belt, but it offers a more modular alternative and a more comprehensive solution to client-side package management. Let's take a look!

5.4.2 Big libraries, small components

Huge libraries such as jQuery do everything you need, as well as things you don't need. For instance, you might not need the animations or the AJAX that come with it. In this sense, struggling to keep pieces out of jQuery using custom builds is an uphill battle; automating the process isn't trivial, and you're doing more to get less, which I guess is what the "write less, do more" slogan refers to.

Component is a tool that's all about small components that do one thing only but do it well. Rather than using a big library for all your needs, TJ Holowaychuk,[7] prolific open source author, advocates using multiple small blocks to build exactly what you need in a modular way and without any added bloat.

The first thing you'll need to do, as usual, is install the CLI tool from `npm`:

```
npm install -g component
```

If you're consuming components, you can get away with a manifest with the bare minimum valid JSON. Let's create that, too:

```
echo "{}" > component.json
```

Installing components such as Lo-Dash works similarly to what you did previously with Bower. The main difference is that rather than using a registry whose sole purpose is tracking packages, like Bower does, Component uses GitHub as its default registry. Specifying the username and repository, as shown in the following command, is enough to fetch a component:

```
component install lodash/lodash
```

In contrast with what other libraries do, Component will always update the manifest, adding the packages you install. You must also add the entry point to the scripts field in the component manifest.

```
"scripts": ["js/app/app.js"]
```

Another difference you can find in Component is that it has an additional build step, which will bundle any components you've installed into a single `build.js` concate-

[7] Read an introduction to Component on Holowaychuk's blog at http://bevacqua.io/bf/component.

nated file. Given that components use CommonJS-style `require` calls, the necessary `require` function will also be provided.

```
component build
```

I encourage you to look at a pair of accompanying samples, which might help you learn how to use Component. The first one, ch05/14_adopting-component, is a fully working example of what has been described here.

The second, ch05/15_automate-component-build, explains how to automate the build step with Grunt, using the `grunt-component-build` package. Such a build step is particularly useful if your code is also treated as components.

To wrap things up, I'll give you an overview of each of the systems we've discussed, which might help you decide on a package manager or module system.

5.4.3 Choosing the right module system

Component has the right idea behind it—modular pieces of code that do one thing well—but it has subtle drawbacks, as well. For instance, it has an unnecessary build step in `component install`. Executing `component install` should build everything you need for the components to work, the way `npm` does. It's also kind of mystical to configure, and the documentation is hard to find. Poor naming is a huge drawback in this regard, as you can't do a web search for Component and not get unrelated results, making it hard to find the documentation you want.

Bower is fine if you don't buy into the CJS concept, and it's certainly better than downloading code and placing it into directories by yourself and dealing with version upgrades on your own. Bower is fine for fetching packages, but it does little to help you with modularity, and that's where it falls short.

As far as Browserify goes, at the moment it's the best option that's available to us, if you're willing to concede that CJS is the simplest module format available today. The lack of a package manager embedded into Browserify is a good thing, because it doesn't matter which source you pick for modules you consume. They can come from npm, Bower, GitHub, or somewhere else.

Browserify provides mechanisms for both bringing vendor code into the CJS format and exporting a CJS formatted application into a single file. As we discussed in 5.3.3, Browserify can produce source maps that help debug during development, and using it gives you access to any CJS modules originally written for Node development.

Last, AMD modules might be a good fit for using Bower, because they don't interfere with each other. The benefit here is that you don't have to learn the CJS approach, although I would argue that there isn't all that much to learn about it.

Before discussing the changes coming to the JavaScript language in ECMAScript 6, there's one more topic we need to tend to. That's the topic of circular dependencies, such as a chicken depending on an egg that depends on a chicken.

5.4.4 Learning about circular dependencies

Circular dependencies, explained previously as a chicken depending on an egg that depends on a chicken, are a tough nut to crack, and they're straight up unsupported by many module systems. In this brief section I aim to dispel any issues you have by answering the following questions:

- Is there a good reason to use circular dependencies?
- What patterns can you use to avoid them?
- How do the solutions we've talked about handle circular dependencies?

Components that depend on each other represent a code smell, meaning there might be a deeper problem in your code. The best approach to circular dependencies is to avoid them altogether. You can use a few patterns to avoid them. If two components are talking to each other, it might be a sign that they need to communicate through a service they both consume, for example. That way, it'll be easier to reason about (and write code for) the affected components. In chapter 7, you'll look at the ways you can avoid these chicken-and-egg type of situations when using AngularJS in client-side applications.

Using a service as a middleman is one of many ways to solve circular dependencies. You might have your `chicken` module depend on `egg` and talk to it directly, but if `egg` wants to talk to `chicken`, then it should use the callbacks `chicken` gives to it. An even simpler approach is to have instances of your modules depend on each other. Have a `chicken` and an `egg` depending on each other, rather than the entire families, and the problem is circumvented.

You also need to take into account that different systems deal with circular dependencies differently. If you try to resolve a circular dependency in Angular, it will throw an error. Angular doesn't provide any mechanisms to deal with circular dependencies at the module level. You can get around this by using their dependency resolver. Once an `egg` module that depends on the `chicken` module is resolved, then the `chicken` module can fetch the `egg` module when it's used.

In the case of AMD modules, if you define a circular dependency such that `chicken` needs `egg` and `egg` needs `chicken`, then when `egg`'s module function is called, it will get an `undefined` value for `chicken`. `egg` can fetch `chicken` later, after modules have been defined by using the `require` method.

CommonJS allows circular dependencies by pausing module resolution whenever a `require` call is made. If a `chicken` module requires an `egg` module, then interpretation of the `chicken` module is halted. When the `egg` module requires `chicken`, it will get the partial representation of the `chicken` module, until the `require` call is made. Then the `chicken` module will finish being interpreted. The code sample labeled ch05/16_circular-dependencies illustrates this point.

The bottom line is that you should avoid circular dependencies like the plague. Circular dependencies introduce unnecessary complexity into your programs,

module systems don't have a standard way of dealing with them, and they can always be avoided by writing code in a more organized way.

To wrap up this chapter, we'll go through a few changes coming to the language in ECMAScript 6, and what they bring to the table when it comes to modular component design.

5.5 Harmony: a glimpse of ECMAScript 6

As you might know, ECMAScript (ES) is the spec that defines the behavior of JavaScript code. ES6, also known as Harmony, is the (long-awaited) upcoming version of the spec. Once ES6 lands, you'll benefit from hundreds of small and large improvements to the language, part of which I'll cover in this section. At the time of this writing, parts of Harmony are in Chrome Canary, the edge version of Google Chrome, and also in the Firefox Nightly build. In Node, you can use the `--harmony` flag when invoking the `node` process to enable ES6 language features.

Please note that ES6 features are highly experimental and subject to change; the spec is constantly in flux. Take what's discussed in this section with a pinch of salt. I'll introduce you to concepts and syntax in the upcoming language release; features proposed as part of ES6 at this point are unlikely to change, but specific syntax is more likely to be tweaked.

Google has made an interesting effort in popularizing ES6 learning through their Traceur project, which compiles ES6 down to ES3 (a generally available spec version), allowing you to write code in ES6 and then execute the resulting ES3. Although Traceur doesn't support every feature in Harmony, it's one of the most featured compilers available.

5.5.1 Traceur as a Grunt task

Traceur is available as a Grunt task, thanks to a package called `grunt-traceur`. You can use the following configuration to set it up. It will compile each file individually and place the results in a `build` directory:

```
traceur: {
  build: {
    src: 'js/**/*.js',
    dest: 'build/'
  }
}
```

With the help of this task, you can compile a few of the ES6 Harmony examples I'll show you along the way. Naturally, the accompanying code samples have a working example of this Grunt task, as well as a few different snippets of what you can do with Harmony, so be sure to check out `ch05/17_harmony-traceur` and skim through those samples. Chapters 6 and 7 also contain more pieces of ES6 code, to give you a better picture of what features are coming to the language.

Now that you know of a few ways to turn ES6 features on, let's dive into Harmony's way of doing modules.

5.5.2 Modules in Harmony

Throughout this chapter, you've navigated different module systems and learned about modular design patterns. Input from both AMD and CJS have influenced the design decisions behind Harmony modules, in a way that aims to please proponents of either system. These modules have their own scope; they export public API members using the `export` keyword, which can later be imported individually using the `import` keyword. An optional explicit `module` declaration allows for file concatenation.

What follows is an example of how these mechanics work. I'm using the latest syntax available[8] at the time of this writing. The syntax comes from a meeting held in March 2013 by TC39, the technical committee in charge of moving the language forward. If I were you, I wouldn't focus too much on the specifics, only the general idea.

To begin with, you'll define a basic module with a couple of exported methods:

```
// math.js

export var pi = 3.141592;

export function circumference (radius) {
    return  2 * pi * radius;
}
```

Consuming these methods is a matter of referencing them in an `import` statement, as shown in the following code snippet. These statements can choose to import one, many, or all the exports found in a module. The following statement imports the `circumference` export into the local module:

```
import { circumference } from "math";
```

If you want to import multiple exports, you comma-separate them:

```
import { circumference, pi } from "math";
```

Importing every export from a module in an object, rather than directly on the local context, can be done using the `as` syntax:

```
import "math" as math;
```

If you want to define modules explicitly, rather than having them be defined implicitly, for release scenarios where you're going to bundle your scripts in a single file, there's a literal way in which you can define a module:

```
module "math" {
    export // etc...
};
```

If you're interested in the module system in ES6, you should read an article[9] that encompasses what you've learned so far about ES6, and sheds light on the module

[8] Find the ES6 article at http://bevacqua.io/bf/es6-modules.
[9] Find this ES6 article at http://bevacqua.io/bf/es6-modules.

system's extensibility. Always keep in mind that the syntax is subject to change. Before heading to chapter 6, I have one last little ES6 feature to touch on with regard to modularity. That's the `let` keyword.

5.5.3 Let there be block scope

The ES6 `let` keyword is an alternative to `var` statements. You may remember that `var` is function scoped, as you analyzed in section 5.1.3. With `let`, you get block scoping instead, which is more akin to the scoping rules found in traditional languages. Hoisting plays an important role when it comes to variable declaration, and `let` is a great way to get around the limitations of function scoping in certain cases.

Consider, for instance, the scenario below, a typical situation where you conditionally want to declare a variable. Hoisting makes it awkward to declare the variable inside the `if`, because you know it'll get hoisted to the top of the scope, and keeping it inside the `if` block might cause trouble if someday you decide to use the same variable name in the `else` block.

```
function processImage (image, generateThumbnail) {
    var thumbnailService;
    if (generateThumbnail) {
        thumbnailService = getThumbnailService();
        thumbnailService.generate(image);
    }

    return process(image);
}
```

Using the `let` keyword you could get away with declaring it in the `if` block, not worrying about it leaking outside of that block, and without the need to split the variable declaration from its assignment:

```
function processImage (image, generateThumbnail) {
    if (generateThumbnail) {
        let thumbnailService = getThumbnailService();
        thumbnailService.generate(image);
    }

    return process(image);
}
```

The difference is subtle in this case, but getting away from having a long list of variables listed on the top of a function scope, which might only be used in one of the code paths, is a code smell in current JavaScript implementations using `var`. It's a code smell that could easily be avoided by using the `let` keyword, keeping variables in the block scope they belong to.

5.6 Summary

At long last, you're done with scoping, module systems, and so on!

- You learned that keeping code self-contained that has a clear purpose, as well as information hiding, can greatly improve your interface designs.

- Scoping, `this`, and hoisting are much clearer now, which will help you design code that fits the JavaScript paradigm better, without even realizing it.
- Using closures and the module pattern taught you how module systems work.
- You compared how CommonJS, RequireJS, and Angular deal with module loading, and how they handle circular dependencies.
- You learned about the importance of testability, which we'll expand on in chapter 8, and how the Inversion of Control pattern can make your code more testable.
- We discussed how to leverage npm packages in the browser thanks to Browserify, downloading dependencies with Bower, and the UNIX philosophy of writing modular code with Component.
- You saw what's coming in ES6, such as the module system and the `let` keyword, and you learned how to play around with ES6 using the Traceur compiler.

The fate of most applications is often sealed before a single line of code has been written. How is that possible? Simply, bad design assures bad results. Good design and effective processes are the foundation on which maintainable applications are built, scaled, and improved. For JavaScript developers, this means discovering the tooling, modern libraries, and architectural patterns that enable those improvements.

JavaScript Application Design: A Build First Approach introduces techniques to improve software quality and development workflow. You'll begin by learning how to establish processes designed to optimize the quality of your work. You'll execute tasks whenever your code changes, run tests on every commit, and deploy in an automated fashion. Then you'll focus on designing modular components and composing them together to build robust applications.

What's inside

- Automated development, testing, and deployment processes
- JavaScript fundamentals and modularity best practices
- Modular, maintainable, and well-tested applications
- Master asynchronous flows, embrace MVC, and design a REST API

This book assumes readers understand the basics of JavaScript.

Dealing with collections

As applications grow in size, the amount of data they have to deal with often grows as well. Learning how to manage and manipulate collections of data is essential. The following outstanding chapter takes a deep dive into working with JavaScript collections and includes two new ES6 collections: maps and sets.

Chapter 9 from *Secrets of the JavaScript Ninja, Second Edition*
by John Resig, Bear Bibeault, and Josip Maras

Dealing with collections

This chapter covers

- Creating and modifying arrays
- Using and reusing array functions
- Creating dictionaries with maps
- Creating collections of unique objects with sets

Now that we've spent some time wrangling the particularities of object-orientation in JavaScript, we'll move on to a closely related topic: collections of items. We'll start with arrays, the most basic type of collection in JavaScript, and look at some array peculiarities you may not expect if your programming background is in another programming language. We'll continue by exploring some of the built-in array methods that will help you write more elegant array-handling code.

Next, we'll discuss two new ES6 collections: maps and sets. Using maps, you can create dictionaries of a sort that carry mappings between keys and values—a collection that's extremely useful in certain programming tasks. Sets, on the other hand, are collections of unique items in which each item can't occur more than once.

Let's begin our exploration with the simplest and most common of all collections: arrays.

Do you know?

- What are some of the common pitfalls of using objects as dictionaries or maps?
- What value types can a key/value pair have in a `Map`?
- Must the items in a `Set` be of the same type?

9.1 Arrays

Arrays are one of the most common data types. Using them, you can handle collections of items. If your programming background is in a strongly typed language such as C, you probably think of arrays as sequential chunks of memory that house items of the same type, where each chunk of memory is of fixed size and has an associated index through which you can easily access it.

But as with many things in JavaScript, arrays come with a twist: They're just objects. Although this leads to some unfortunate side effects, primarily in terms of performance, it also has some benefits. For example, arrays can access methods, like other objects—methods that will make our lives a lot easier.

In this section, we'll first look at ways to create arrays. Then we'll explore how to add items to and remove items from different positions in an array. Finally, we'll examine the built-in array methods that will make our array-handling code much more elegant.

9.1.1 Creating arrays

There are two fundamental ways to create new arrays:

- Using the built-in `Array` constructor
- Using array literals `[]`

Let's start with a simple example in which we create an array of ninjas and an array of samurai.

Listing 9.1 Creating arrays

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = new Array("Oda", "Tomoe");

assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");

assert(ninjas[0] === "Kuma", "Kuma is the first ninja");
assert(samurai[samurai.length-1] === "Tomoe",
       "Tomoe is the last samurai");

assert(ninjas[4] === undefined,
      "We get undefined if we try to access an out of bounds index");
```

To create an array, we can use an array literal [] …

… or the built-in Array constructor.

The length property tells us the size of the array.

We access array items with index notation: The first item is indexed with 0, and the last with array.length – 1.

Reading items outside the array bounds results in "undefined".

```
ninjas[4] = "Ishi";
assert(ninjas.length === 5,
       "Arrays are automatically expanded");
```

Writing to indexes outside the array bounds extends the array.

```
ninjas.length = 2;
assert(ninjas.length === 2, "There are only two ninjas now");
assert(ninjas[0] === "Kuma" && ninjas[1] === "Hattori",
        "Kuma and Hattori");
assert(ninjas[2] === undefined, "But we've lost Yagyu");
```

Manually overriding the length property with a lower value deletes the excess items.

In listing 9.1, we start by creating two arrays. The `ninjas` array is created with a simple array literal:

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
```

It's immediately prefilled with three ninjas: Kuma, Hattori, and Yagyu. The `samurai` array is created using the built-in `Array` constructor:

```
const samurai = new Array("Oda", "Tomoe");
```

Array literals vs. the Array constructor

Using array literals to create arrays is preferred over creating arrays with the `Array` constructor. The primary reason is simplicity: `[]` versus `new Array()` (2 characters versus 11 characters—hardly a fair contest). In addition, because JavaScript is highly dynamic, nothing stops someone from overriding the built-in `Array` constructor, which means calling `new Array()` doesn't necessarily have to create an array. Thus we recommend that you generally stick to array literals.

Regardless of how we create it, each array has a `length` property that specifies the size of the array. For example, the `length` of the `ninjas` array is `3`, and it contains 3 ninjas. We can test this with the following assertions:

```
assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");
```

As you probably know, you access array items by using index notation, where the first item is positioned at index `0` and the last item at index `array.length – 1`. But if we try to access an index outside those bounds—for example, with `ninjas[4]` (remember, we have only three ninjas!), we won't get the scary "Array index out of bounds" exception that we receive in most other programming languages. Instead, `undefined` is returned, signaling that there's nothing there:

```
assert(ninjas[4] === undefined,
       "We get undefined if we try to access an out of bounds index");
```

This behavior is a consequence of the fact that JavaScript arrays are objects. Just as we'd get `undefined` if we tried to access a nonexistent object property, we get `undefined` when accessing a nonexistent array index.

On the other hand, if we try to write to a position outside of array bounds, as in

```
ninjas[4] = "Ishi";
```

the array will expand to accommodate the new situation. For example, see figure 9.1: We've essentially created a hole in the array, and the item at index `3` is `undefined`. This also changes the value of the `length` property, which now reports a value of `5`, even though one array item is `undefined`.

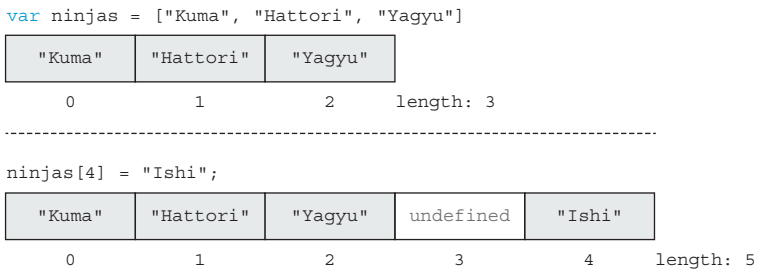


Figure 9.1 Writing to an array index outside of array bounds expands the array.

Unlike in most other languages, in JavaScript, arrays also exhibit a peculiar feature related to the `length` property: Nothing stops us from manually changing its value. Setting a value higher than the current length will expand the array with `undefined` items, whereas setting the value to a lower value will trim the array, as in `ninjas.length = 2;`.

Now that we've gone through the basics of array creation, let's go through some of the most common array methods.

9.1.2 Adding and removing items at either end of an array

Let's start with the following simple methods we can use to add items to and remove items from an array:

- `push` adds an item to the end of the array.
- `unshift` adds an item to the beginning of the array.
- `pop` removes an item from the end of the array.
- `shift` removes an item from the beginning of the array.

You've probably already used these methods, but just in case, let's make sure we're on the same page by exploring the following listing.

Listing 9.2 Adding and removing array items

```
const ninjas = [];
assert(ninjas.length === 0, "An array starts empty");

ninjas.push("Kuma");
assert(ninjas[0] === "Kuma",
       "Kuma is the first item in the array");
assert(ninjas.length === 1, "We have one item in the array");

ninjas.push("Hattori");
assert(ninjas[0] === "Kuma",
       "Kuma is still first");
assert(ninjas[1] === "Hattori",
       "Hattori is added to the end of the array");
assert(ninjas.length === 2,
       "We have two items in the array!");

ninjas.unshift("Yagyu");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is the first item");
assert(ninjas[1] === "Kuma",
       "Kuma moved to the second place");
assert(ninjas[2] === "Hattori",
       "And Hattori to the third place");
assert(ninjas.length === 3,
       "We have three items in the array!");

const lastNinja = ninjas.pop();
assert(lastNinja === "Hattori",
       "We've removed Hattori from the end of the array");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is still the first item");
assert(ninjas[1] === "Kuma,
       "Kuma is still in second place");
assert(ninjas.length === 2,
       "Now there are two items in the array");

const firstNinja = ninjas.shift();
assert(firstNinja === "Yagyu",
       "We've removed Yagyu from the beginning of the array");
assert(ninjas[0] === "Kuma",
       "Kuma has shifted to the first place");
assert(ninjas.length === 1,
       "There's only one ninja in the array");
```

Creates a new, empty array.

Pushes a new item to the end of the array.

Pushes another item to the end of the array.

Uses the built-in unshift method to insert the item at the beginning of the array. Other items are adjusted accordingly.

Pops the last item from the array.

Removes the first item from the array. Other items are moved to the left accordingly.

In this example, we first create a new, empty `ninjas` array:

```
ninjas = [] // ninjas: []
```

In each array, we can use the built-in `push` method to append an item to the end of the array, changing its length in the process:

```
ninjas.push("Kuma"); // ninjas: ["Kuma"];
ninjas.push("Hattori"); // ninjas: ["Kuma", "Hattori"];
```

We can also add new items to the beginning of the array by using the built in `unshift` method:

```
ninjas.unshift("Yagyu");// ninjas: ["Yagyu", "Kuma", "Hattori"];
```

Notice how existing array items are adjusted. For example, before calling the `unshift` method, `"Kuma"` was at index `0`, and afterward it's at index `1`.

We can also remove elements from either the end or the beginning of the array. Calling the built-in `pop` method removes an element from the end of the array, reducing the array's length in the process:

```
var lastNinja = ninjas.pop(); // ninjas:["Yagyu", "Kuma"]
                              // lastNinja: "Hattori"
```

We can also remove an item from the beginning of the array by using the built-in `shift` method:

```
var firstNinja = ninjas.shift(); //ninjas: ["Kuma"]
                                 //firstNinja: "Yagyu"
```

Figure 9.2 shows how `push`, `pop`, `shift`, and `unshift` modify arrays.

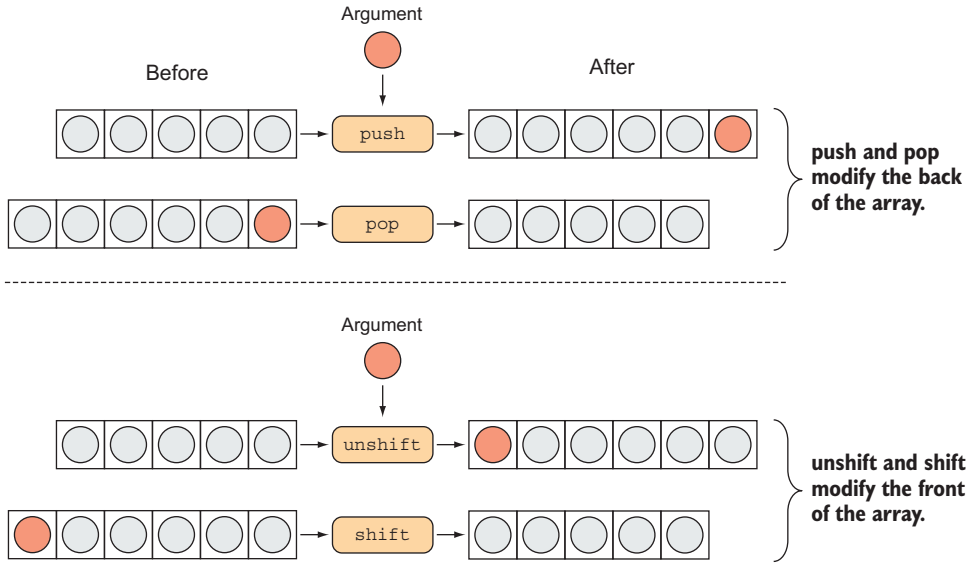


Figure 9.2 The `push` and `pop` methods modify the end of an array, whereas `shift` and `unshift` modify the array's beginning.

Performance considerations: `pop` and `push` vs. `shift` and `unshift`

The `pop` and `push` methods only affect the last item in an array: `pop` by removing the last item, and `push` by inserting an item at the end of the array. On the other hand, the `shift` and `unshift` methods change the first item in the array. This means the indexes of any subsequent array items have to be adjusted. For this reason, `push` and `pop` are significantly faster operations than `shift` and `unshift`, and we recommend using them unless you have a good reason to do otherwise.

9.1.3 Adding and removing items at any array location

The previous example removed items from the beginning and end of the array. But this is too constraining—in general, we should be able to remove items from any array location. One straightforward approach for doing this is shown in the following listing.

Listing 9.3 Naïve way to remove an array item

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

delete ninjas[1];                    <-- Uses the delete command to delete an item.

assert(ninjas.length === 4,
       "Length still reports that there are 4 items");

assert(ninjas[0] === "Yagyu", "First item is Yagyu");
assert(ninjas[1] === undefined, "We've simply created a hole");
assert(ninjas[2] === "Hattori", "Hattori is still the third item");
assert(ninjas[3] === "Fuma", "And Fuma is the last item");
```

We deleted an item, but the array still reports that it has 4 items. We've only created a hole in the array.

This approach to deleting an item from an array doesn't work. We effectively only create a hole in the array. The array still reports that it has four items, but one of them—the one we wanted to delete—is `undefined` (see figure 9.3).

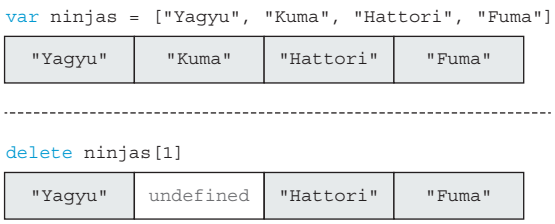


Figure 9.3 Deleting an item from an array creates a hole in the array.

Similarly, if we wanted to insert an item at an arbitrary position, where would we even start? As an answer to these problems, all JavaScript arrays have access to the `splice` method: Starting from a given index, this method removes and inserts items. Check out the following example.

Listing 9.4 Removing and adding items at arbitrary positions

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

var removedItems = ninjas.splice(1, 1);

assert(removedItems.length === 1, "One item was removed");
assert(removedItems[0] === "Kuma");

assert(ninjas.length === 3,
       "There are now three items in the array");
assert(ninjas[0] === "Yagyu",
       "The first item is still Yagyu");
assert(ninjas[1] === "Hattori",
       "Hattori is now in the second place");
assert(ninjas[2] === "Fuma",
       "And Fuma is in the third place");

removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
assert(removedItems.length === 2, "Now, we've removed two items");
   assert(removedItems[0] === "Hattori", "Hattori was removed");
assert(removedItems[1] === "Fuma", "Fuma was removed");
assert(ninjas.length === 4, "We've inserted some new items");
assert(ninjas[0] === "Yagyu", "Yagyu is still here");
assert(ninjas[1] === "Mochizuki", "Mochizuki also");
assert(ninjas[2] === "Yoshi", "Yoshi also");
assert(ninjas[3] === "Momochi", "and Momochi");
```

Creates a new array with four items.

Uses the built-in splice method to remove one element, starting at index I.

splice returns an array of the removed items. In this case, we removed one item.

The ninja array no longer contains Kuma; subsequent items were automatically shifted.

We can insert an element at a position by adding arguments to the splice call.

We begin by creating a new array with four items:

```
var ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];
```

Then we call the built-in `splice` method:

```
var removedItems = ninjas.splice(1,1);//ninjas:["Yagyu","Hattori", "Fuma"];
                                      //removedItems: ["Kuma"]
```

In this case, `splice` takes two arguments: the index from which the splicing starts, and the number of elements to be removed (if we leave out this argument, all elements to the end of the array are removed). In this case, the element with index `1` is removed from the array, and all subsequent elements are shifted accordingly.

In addition, the `splice` method returns an array of items that have been removed. In this case, the result is an array with a single item: `"Kuma"`.

Using the `splice` method, we can also insert items into arbitrary positions in an array. For example, consider the following code:

```
removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
         //ninjas: ["Yagyu", "Mochizuki", "Yoshi", "Momochi"]
//removedItems: ["Hattori", "Fuma"]
```

Starting from index `1`, it first removes two items and then adds three items: `"Mochizuki"`, `"Yoshi"`, and `"Momochi"`.

Now that we've given you a refresher on how arrays work, let's continue by studying some common operations that are often performed on arrays. These will help you write more elegant array-handling code.

9.1.4 Common operations on arrays

In this section, we'll explore some of the most common operations on arrays:

- *Iterating* (or *traversing*) through arrays
- *Mapping* existing array items to create a new array based on them
- *Testing* array items to check whether they satisfy certain conditions
- *Finding* specific array items
- *Aggregating* arrays and computing a single value based on array items (for example, calculating the sum of an array)

We'll start with the basics: array iterations.

ITERATING OVER ARRAYS

One of the most common operations is iterating over an array. Going back to Computer Science 101, an iteration is most often performed in the following way:

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

for(let i = 0; i < ninjas.length; i++){
  assert(ninjas[i] !== null, ninjas[i]);
}
```

Reports the value of each ninja

This example is as simple as it looks. It uses a `for` loop to check every item in the array; the results are shown in figure 9.4.

You've probably written something like this so many times that you don't even have to think about it anymore. But just in case, let's take a closer look at the `for` loop.

To go through an array, we have to set up a counter variable, `i`, specify the number up to which we want to count (`ninjas.length`), and

Figure 9.4 The output of checking the ninjas with a `for` loop

define how the counter will be modified (`i++`). That's an awful lot of bookkeeping to perform such a common action, and it can be a source of annoying little bugs. In addition, it makes our code more difficult to read. Readers have to look closely at every part of the `for` loop, just to be sure it goes through all the items and doesn't skip any.

To make life easier, all JavaScript arrays have a built-in `forEach` method we can use in such situations. Look at the following example.

Listing 9.5 Using the `forEach` method

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

ninjas.forEach(ninja => {
  assert(ninja !== null, ninja);
));
```

Uses the built-in forEach method to iterate over the array

We provide a callback (in this case, an arrow function) that's called *immediately*, for each item in the array. That's it—no more fussing about the start index, the end condition, or the exact nature of the increment. The JavaScript engine takes care of all that for us, behind the scenes. Notice how much easier to understand this code is, and how it has fewer bug-spawning points.

We'll continue by taking things up a notch and seeing how we can map arrays to other arrays.

MAPPING ARRAYS

Imagine that you have an array of `ninja` objects. Each ninja has a name and a favorite weapon, and you want to extract an array of weapons from the `ninjas` array. Armed with the knowledge of the `forEach` method, you might write something like the following listing.

Listing 9.6 Naïve extraction of a weapons array

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = [];
ninjas.forEach(ninja => {
  weapons.push(ninja.weapon);
});

assert(weapons[0] === "shuriken"
    && weapons[1] === "katana"
    && weapons[2] === "wakizashi"
    && weapons.length === 3,
    "The new array contains all weapons");
```

Creates a new array and uses a forEach loop over ninjas to extract individual ninja weapons

This isn't all that bad: We create a new, empty array, and use the `forEach` method to iterate over the `ninjas` array. Then, for each `ninja` object, we add the current weapon to the `weapons` array.

As you might imagine, creating new arrays based on the items in an existing array is surprisingly common—so common that it has a special name: *mapping* an array. The idea is that we map each item from one array to a new item of a new array. Conveniently, JavaScript has a `map` function that does exactly that, as shown in the following listing.

Listing 9.7 Mapping an array

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = ninjas.map(ninja => ninja.weapon);

assert(weapons[0] === "shuriken"
    && weapons[1] === "katana"
    && weapons[2] === "wakizashi"
    && weapons.length == 3, "The new array contains all weapons");
```

The built-in map method takes a function that's called for each item in the array.

The built-in `map` method constructs a completely new array and then iterates over the input array. For each item in the input array, `map` places exactly one item in the newly constructed array, based on the result of the callback provided to `map`. The inner workings of the `map` function are shown in figure 9.5.

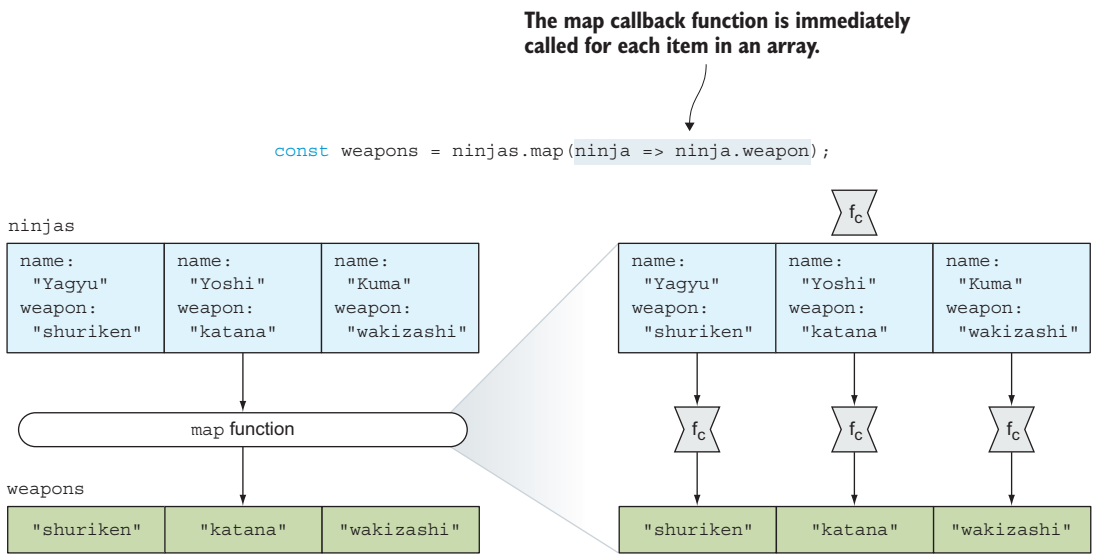


Figure 9.5 The `map` function calls the provided callback function (f_C) on each array item, and creates a new array with callback return values.

Now that we know how to map arrays, let's see how to test array items for certain conditions.

TESTING ARRAY ITEMS

When working with collections of items, we'll often run into situations where we need to know whether *all* or at least *some* of the array items satisfy certain conditions. To write this code as efficiently as possible, all JavaScript arrays have access to the built-in `every` and `some` methods, shown next.

Listing 9.8 Testing arrays with the `every` and `some` methods

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];

const allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
const allNinjasAreArmed = ninjas.every(ninja => "weapon" in ninja);

assert(allNinjasAreNamed, "Every ninja has a name");
assert(!allNinjasAreArmed, "But not every ninja is armed");

const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
assert(someNinjasAreArmed, "But some ninjas are armed");
```

The built-in every method takes a callback that's called for each array item. It returns true if the callback returns a true value for *all* array items, or false otherwise.

The built-in some method also takes a callback. It returns true if the callback returns a true value for at least one array item, or false otherwise.

Listing 9.8 shows an example where we have a collection of `ninja` objects but are unsure of their names and whether all of them are armed. To get to the root of this problem, we first take advantage of `every`:

```
var allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
```

The `every` method takes a callback that, for each ninja in the collection, checks whether we know the ninja's name. `every` returns `true` only if the passed-in callback returns `true` for *every* item in the array. Figure 9.6 shows how `every` works.

In other cases, we only care whether *some* array items satisfy a certain condition. For these situations, we can use the built-in method `some`:

```
const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
```

Starting from the first array item, `some` calls the callback on each array item until an item is found for which the callback returns a `true` value. If such an item is found, the return value is `true`; if not, the return value is `false`.

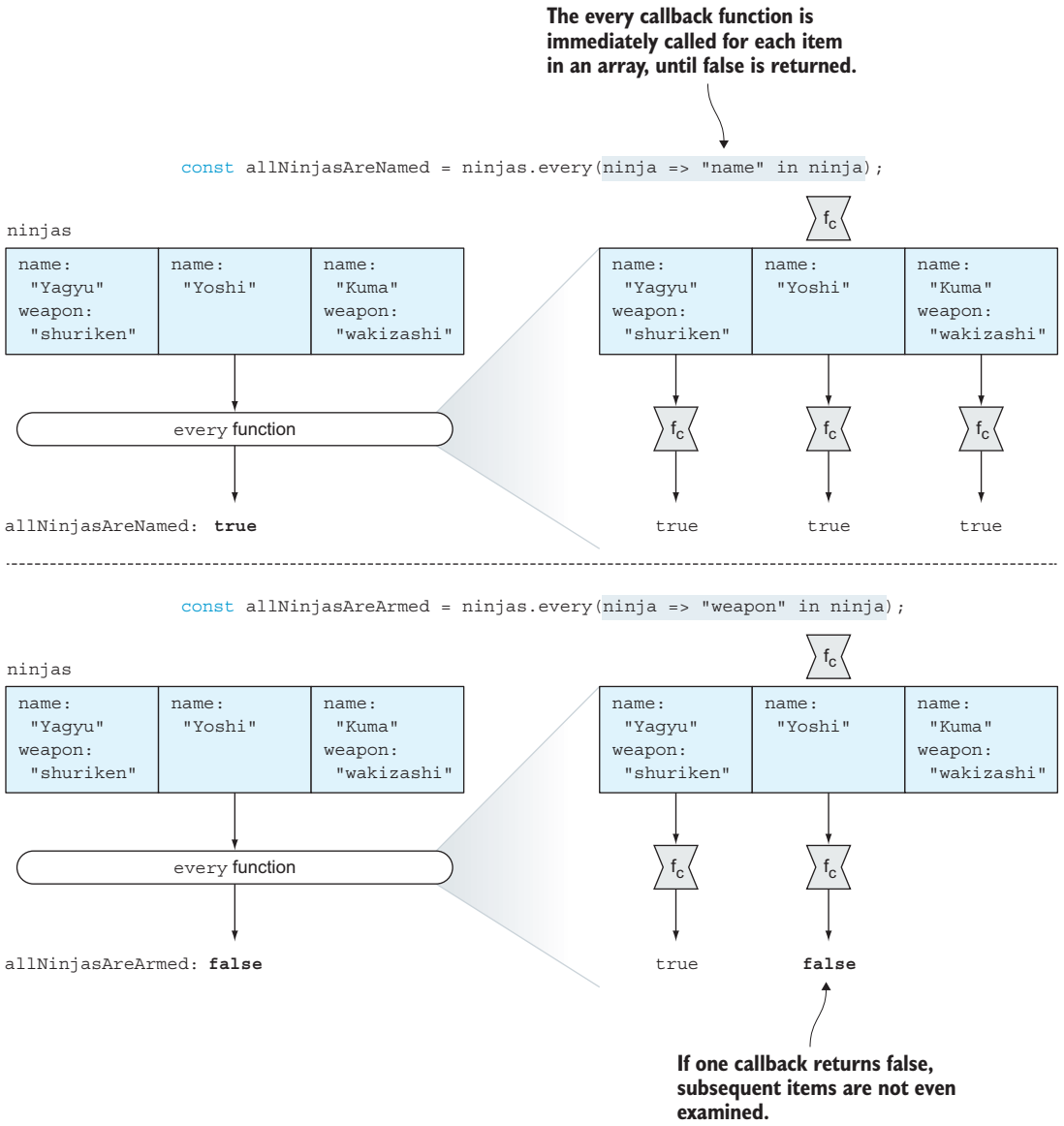


Figure 9.6 The `every` method tests whether all items in an array satisfy a certain condition represented by a callback.

Figure 9.7 shows how `some` works under the hood: We search an array in order to find out whether some or all of its items satisfy a certain condition.

Next we'll explore how to search an array to find a particular item.

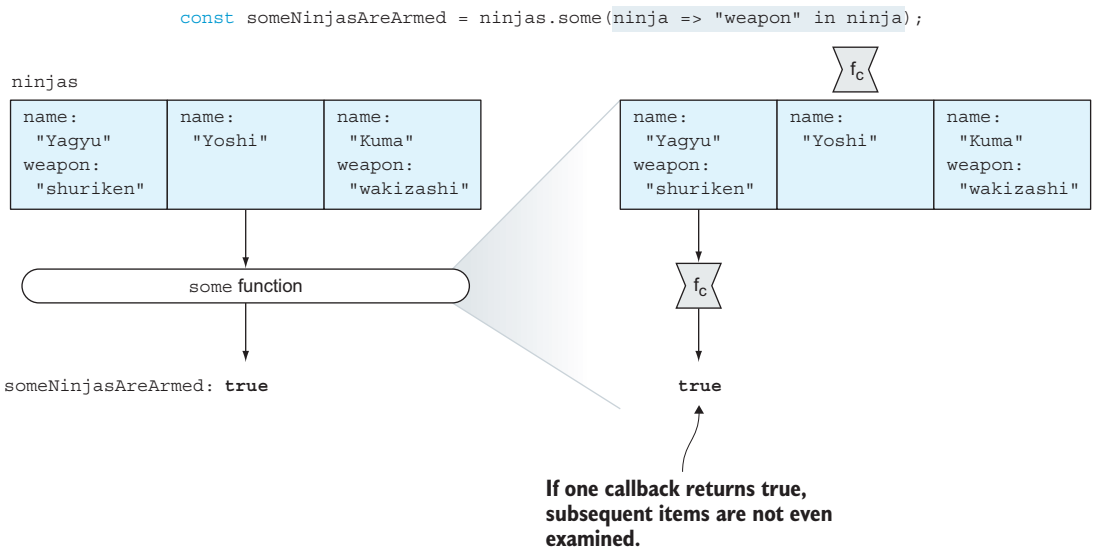


Figure 9.7 The `some` method checks whether at least one array item satisfies a condition represented by the passed-in callback.

SEARCHING ARRAYS

Another common operation that you're bound to use, sooner rather than later, is finding items in an array. Again, this task is greatly simplified with another built-in array method: `find`. Let's study the following listing.

NOTE The built-in `find` method is part of the ES6 standard. For current browser compatibility, see http://mng.bz/U532.

Listing 9.9 Finding array items

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];

const ninjaWithWakizashi = ninjas.find(ninja => {
  return ninja.weapon === "wakizashi";
});

assert(ninjaWithWakizashi.name === "Kuma"
    && ninjaWithWakizashi.weapon === "wakizashi",
    "Kuma is wielding a wakizashi");
```

Uses the find method to find the first array item that satisfies a certain condition, represented by a passed-in callback.

```
const ninjaWithKatana = ninjas.find(nina => {
  return ninja.weapon === "katana";
});

assert(ninjaWithKatana === undefined,
      "We couldn't find a ninja that wields a katana");

const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);

assert(armedNinjas.length === 2, "There are two armed ninjas:");
assert(armedNinjas[0].name === "Yagyu"
    && armedNinjas[1].name === "Kuma", "Yagyu and Kuma");
```

The find method returns undefined if an item can't be found.

Use the filter method to find multiple items that all satisfy a certain condition.

It's easy to find an array item that satisfies a certain condition: We use the built-in `find` method, passing it a callback that's invoked for each item in the collection until the targeted item is found. This is indicated by the callback returning `true`. For example, the expression

```
ninjas.find(ninja => ninja.weapon === "wakizashi");
```

finds Kuma, the first ninja in the `ninjas` array that's wielding a wakizashi.

If we've gone through the entire array without a single item returning `true`, the final result of the search is `undefined`. For example, the code

```
ninjaWithKatana = ninjas.find(ninja => ninja.weapon === "katana");
```

returns `undefined`, because there isn't a katana-wielding ninja. Figure 9.8 shows the inner workings of the `find` function.

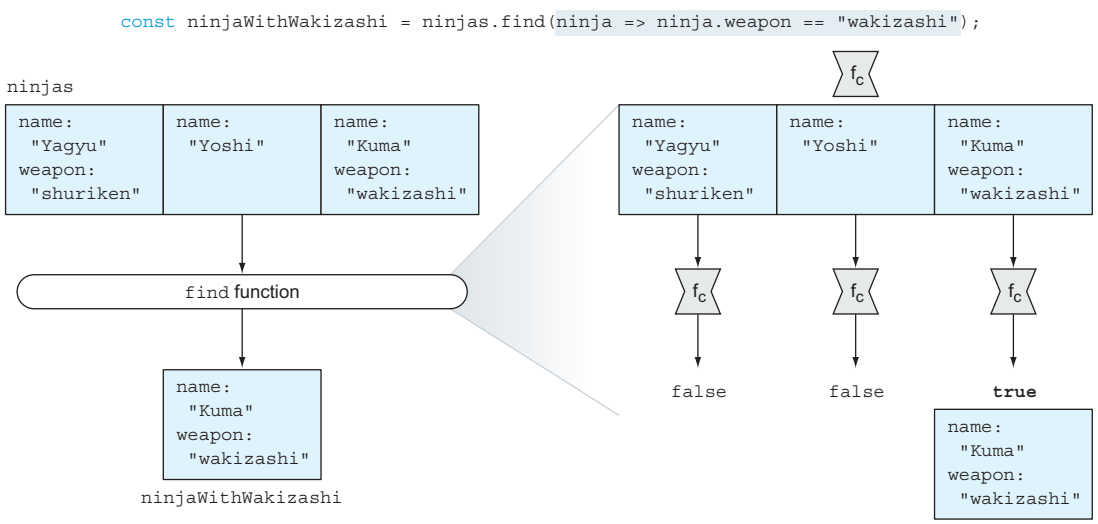


Figure 9.8 The `find` function finds one item in an array: the first item for which the `find` callback returns `true`.

If we need to find multiple items satisfying a certain criterion, we can use the `filter` method, which creates a *new* array containing all the items that satisfy that criterion. For example, the expression

```
const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);
```

creates a new `armedNinjas` array that contains only `ninjas` with a weapon. In this case, poor unarmed Yoshi is left out. Figure 9.9 shows how the `filter` function works.

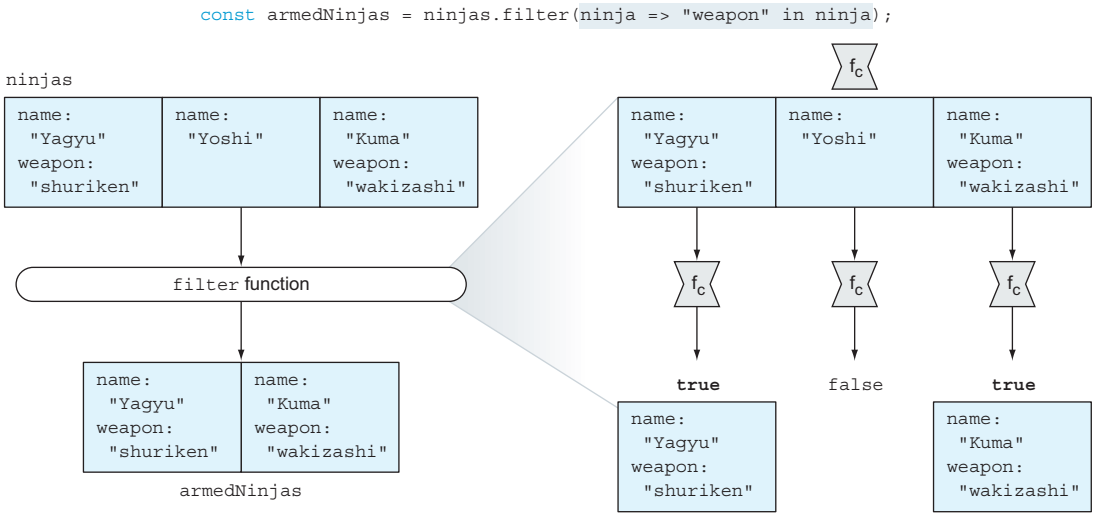


Figure 9.9 The `filter` function creates a new array that contains all items for which the callback returns `true`.

Throughout this example, you've seen how to find particular items in an array, but in many cases it might also be necessary to find the index of an item. Let's take a closer look, with the following example.

Listing 9.10 Finding array indexes

```
const ninjas = ["Yagyu", "Yoshi", "Kuma", "Yoshi"];

assert(ninjas.indexOf("Yoshi") === 1, "Yoshi is at index 1");
assert(ninjas.lastIndexOf("Yoshi") === 3, "and at index 3");

const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");

assert(yoshiIndex === 1, "Yoshi is still at index 1");
```

To find the index of a particular item, we use the built-in `indexOf` method, passing it the item whose index we want to find:

```
ninjas.indexOf("Yoshi")
```

In cases where a particular item can be found multiple times in an array (as is the case with `"Yoshi"` and the `ninjas` array), we may also be interested in finding the last index where Yoshi appears. For this, we can use the `lastIndexOf` method:

```
ninjas.lastIndexOf("Yoshi")
```

Finally, in the most-general case, when we don't have a reference to the exact item whose index we want to search for, we can use the `findIndex` method:

```
const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");
```

The `findIndex` method takes a callback and returns the index of the first item for which the callback returns `true`. In essence, it works a lot like the `find` method, the only difference being that `find` returns a particular item, whereas `findIndex` returns the index of that item.

SORTING ARRAYS

One of the most common array operations is *sorting*—arranging items systematically in some order. Unfortunately, correctly implementing sorting algorithms isn't the easiest of programming tasks: We have to select the best sorting algorithm for the task, implement it, and tailor it to our needs, while, as always, being careful not to introduce subtle bugs. To get this burden off our back, as you saw in chapter 3, all JavaScript arrays have access to the built-in `sort` method, whose usage looks something like this:

```
array.sort((a, b) => a - b);
```

The JavaScript engine implements the sorting algorithm. The only thing we have to provide is a callback that informs the sorting algorithm about the relationship between two array items. The possible results are as follows:

- If a callback returns a value less than `0`, then item `a` should come before item `b`.
- If a callback returns a value equal to `0`, then items `a` and `b` are on equal footing (as far as the sorting algorithm is concerned, they're equal).
- If a callback returns a value greater than `0`, then item `a` should come after item `b`.

Figure 9.10 shows the decisions made by the sorting algorithm depending on the callback return value.

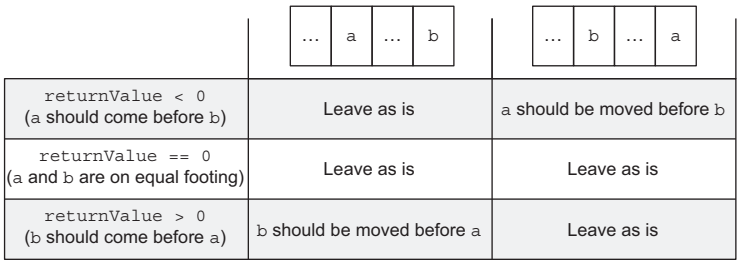


Figure 9.10 If the callback returns a value less than 0, the first item should come before the second. If the callback returns 0, both items should be left as is. And if the return value is greater than 0, the first item should come after the second item.

And that's about all you need to know about the sorting algorithm. The actual sorting is performed behind the scenes, without us having to manually move array items around. Let's look at a simple example.

Listing 9.11 Sorting an array

```
const ninjas = [{name: "Yoshi"}, {name: "Yagyu"}, {name: "Kuma"}];

ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }
  if(ninja1.name > ninja2.name) { return 1; }

  return 0;
});

assert(ninjas[0].name === "Kuma", "Kuma is first");
assert(ninjas[1].name === "Yagyu", "Yagyu is second");
assert(ninjas[2].name === "Yoshi", "Yoshi is third");
```

Passes a callback to the built-in sort method to specify a sorting order

In listing 9.11 we have an array of `ninja` objects, where each ninja has a name. Our goal is to sort that array lexicographically (in alphabetical order), according to ninja names. For this, we naturally use the `sort` function:

```
ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }
  if(ninja1.name > ninja2.name) { return 1; }

  return 0;
});
```

To the `sort` function we only need to pass a callback that's used to compare two array items. Because we want to make a lexical comparison, we state that if `ninja1`'s name is "less" than `ninja2`'s name, the callback returns `-1` (remember, this means `ninja1` should come before `ninja2`, in the final sorted order); if it's greater, the callback returns `1` (`ninja1` should come after `ninja2`); if they're equal, the callback returns `0`. Notice that we can use simple less-than (`<`) and greater-than (`>`) operators to compare two ninja names.

That's about it! The rest of the nitty-gritty details of sorting are left to the JavaScript engine, without us having to worry about them.

AGGREGATING ARRAY ITEMS

How many times have you written code like the following?

```
const numbers = [1, 2, 3, 4];
const sum = 0;

numbers.forEach(number => {
   sum += number;
});

assert(sum === 10, "The sum of first four numbers is 10");
```

This code has to visit every item in a collection and aggregate some value, in essence reducing the entire array to a single value. Don't worry—JavaScript has something to help with this situation, too: the `reduce` method, as shown in the following example.

Listing 9.12 Aggregating items with `reduce`

```
const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((aggregated, number) =>
                             aggregated + number, 0);

assert(sum === 10, "The sum of first four numbers is 10");
```

Uses reduce to accumulate a single value from an array

The `reduce` method works by taking the initial value (in this case, `0`) and calling the callback function on each array item with the result of the previous callback invocation (or the initial value) and the current array item as arguments. The result of the `reduce` invocation is the result of the last callback, called on the last array item. Figure 9.11 sheds more light on the process.

We hope we've convinced you that JavaScript contains some useful array methods that can make our lives significantly easier and your code more elegant, without having to resort to pesky `for` loops. If you'd like to find out more about these and other array methods, we recommend the Mozilla Developer Network explanation at http://mng.bz/cS21.

Now we'll take things a bit further and show you how to reuse these array methods on your own, custom objects.

9.1.5 Reusing built-in array functions

There are times when we may want to create an object that contains a collection of data. If the collection was all we were worried about, we could use an array. But in certain cases, there may be more state to store than just the collection itself—perhaps we need to store some sort of metadata regarding the collected items.

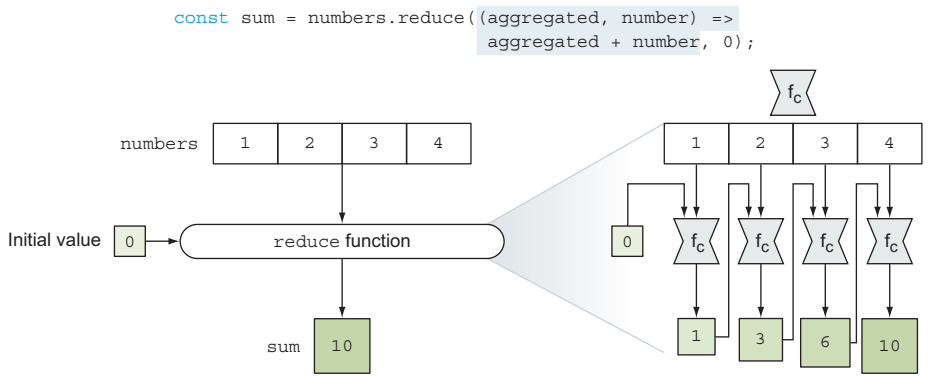


Figure 9.11 The `reduce` function applies a callback to an aggregated value and each item in an array to reduce the array to a single value.

One option may be to create a new array every time you wish to create a new version of such an object, and add the metadata properties and methods to it. Remember, we can add properties and methods to an object as we please, including arrays. Generally, however, this can be slow, not to mention tedious.

Let's examine the possibility of using a normal object and *giving* it the functionality we desire. Methods that know how to deal with collections already exist on the `Array` object; can we trick them into working on our own objects? Turns out that we can, as shown in the following listing.

Listing 9.13 Simulating array-like methods

```
<body>
  <input id="first"/>
  <input id="second"/>
  <script>
    const elems = {
      length: 0,
      add: function(elem){
        Array.prototype.push.call(this, elem);
      },
      gather: function(id){
        this.add(document.getElementById(id));
      },
      find: function(callback){
        return Array.prototype.find.call(this, callback);
      }
    };

    elems.gather("first");
    assert(elems.length === 1 && elems[0].nodeType,
           "Verify that we have an element in our stash");

    elems.gather("second");
    assert(elems.length === 2 && elems[1].nodeType,
           "Verify the other insertion");

    elems.find(elem => elem.id === "second");
    assert(found && found.id === "second",
           "We've found our element");
  </script>
</body>
```

Stores the count of elements. The array needs a place to store the number of items it's storing.

Implements the method to add elements to a collection. The prototype for Array has a method to do this, so why not use it instead of reinventing the wheel?

Implements the gather method to find elements by their id values and add them to the collection.

Implements the method to find elements in the collection. Similar to the add method, it reuses the existing find method accessible to arrays.

In this example, we create a "normal" object and instrument it to mimic some of the behaviors of an array. First we define a `length` property to record the number of elements that are stored, just like an array. Then we define a method to add an element to the end of the simulated array, calling this method `add`:

```
add: function(elem){
  Array.prototype.push.call(this, elem);
}
```

Rather than write our own code, we can use a native method of JavaScript arrays: `Array.prototype.push`.

Normally, the `Array.prototype.push` method would operate on its own array via its function context. But here, we're tricking the method to use *our* object as its context by using the `call` method (remember chapter 4) and forcing our object to be the context of the `push` method. (Notice how we could've just as easily used the `apply` method.) The `push` method, which increments the `length` property (thinking that it's the `length` property of an array), adds a numbered property to the object referencing the passed element. In a way, this behavior is almost subversive (how fitting for ninjas!), but it exemplifies what we can do with mutable object contexts.

The `add` method expects an element reference to be passed for storage. Although sometimes we may have such a reference around, more often than not we won't, so we also define a convenience method, `gather`, that looks up the element by its `id` value and adds it to storage:

```
gather: function(id){
  this.add(document.getElementById(id));
}
```

Finally, we also define a `find` method that lets us find an arbitrary item in our custom object, by taking advantage of the built-in array `find` method:

```
find: function(callback){
  return Array.prototype.find.call(this, callback);
}
```

The borderline nefarious behavior we demonstrated in this section not only reveals the power that malleable function contexts give us, but also shows how we can be clever in reusing code that's already written, instead of constantly reinventing the wheel.

Now that we've spent some time with arrays, let's move on to two new types of collections introduced by ES6: maps and sets.

9.2 Maps

Imagine that you're a developer at freelanceninja.com, a site that wants to cater to a more international audience. For each piece of text on the website—for example, "Ninjas for hire"—you'd like to create a mapping to each targeted language, such as "レンタル用の忍者" in Japanese, "忍者出租" in Chinese, or "?? ??" in Korean (let's hope Google Translate has done an adequate job). These collections, which map a key to a specific value, are called by different names in different programming languages, but most often they're known as *dictionaries* or *maps*.

But how do you efficiently manage this localization in JavaScript? One traditional approach is to take advantage of the fact that objects are collections of named properties and values, and create something like the following dictionary:

```
const dictionary = {
  "ja": {
    "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":"?? ??"
     }
}
assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ");
```

At first glance, this may seem like a perfectly fine approach to this problem, and for this example, it isn't half bad. But unfortunately, in general, you can't rely on it.

9.2.1 Don't use objects as maps

Imagine that somewhere on our site we need to access the translation for the word *constructor*, so we extend the dictionary example into the following code.

Listing 9.14 Objects have access to properties that weren't explicitly defined

```
const dictionary = {
  "ja": {
    "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":"?? ??"
  }
};

assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ",
      "We know how to say 'Ninjas for hire' in Japanese!");

assert(typeof dictionary.ja["constructor"] === "undefined",
       dictionary.ja["constructor"]);
```

We try to access the translation for the word *constructor*—a word that we foolishly forgot to define in our dictionary. Normally, in such a case, we'd expect the dictionary to return `undefined`. But that isn't the result, as you can see in figure 9.12.

As you can see, by accessing the `constructor` property, we obtain the following string:

```
"function Object() { [native code] }"
```

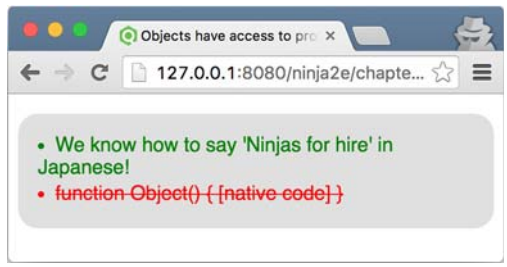


Figure 9.12 Running listing 9.14 shows that objects aren't good maps, because they have access to properties that weren't explicitly defined (through their prototypes).

What's with this?

As you learned in chapter 7, all objects have prototypes; even if we define new, empty objects as our maps, they still have access to the properties of the prototype objects. One of those properties is `constructor` (recall that `constructor` is the property of the prototype object that points back to the constructor function), and it's the culprit behind the mess we now have on our hands.

In addition, with objects, keys can only be string values; if you want to create a mapping for any other value, that value will be silently converted into a string without anyone asking you anything! For example, imagine that we want to track some information about HTML nodes, as in the following listing.

Listing 9.15 Mapping values to HTML nodes with objects

```
<div id="firstElement"></div>
<div id="secondElement"></div>
<script>
  const firstElement = document.getElementById("firstElement");
  const secondElement = document.getElementById("secondElement");

  const map = {};

  map[firstElement] = { data: "firstElement"};
  assert(map[firstElement].data === "firstElement",
     "The first element is correctly mapped");

  map[secondElement] = { data: "secondElement"};
  assert(map[secondElement].data === "secondElement",
     "The second element is correctly mapped");

  assert(map[firstElement].data === "firstElement",
     "But now the firstElement is overriden!");
</script>
```

Defines two HTML elements and fetches them by using the built-in document.getElementById method.

Defines an object that we'll use as a map to store additional information about our HTML elements.

Stores information about the first element, and checks that it was correctly stored.

Stores information about the second element, and checks that it was correctly stored.

The mapping for the first element is now invalid!

In listing 9.15, we create two HTML elements, `firstElement` and `secondElement`, which we then fetch from the DOM by using the `document.getElementById` method. In order to create a mapping that will store additional information about each element, we define a plain old JavaScript object:

```
const map = {};
```

Then we use the HTML element as a key for our mapping object and associate some data with it:

```
map[firstElement] = { data: "firstElement"}
```

And we check that we can retrieve that data. Because that works as it should, we repeat the entire process for the second element:

```
map[secondElement] = { data: "secondElement"};
```

Again, everything looks hunky dory; we've successfully associated some data with our HTML element. But a problem occurs if we decide to revisit the first element:

```
map[firstElement].data
```

It would be normal to assume that we'd again obtain the information about the first element, but this isn't the case. Instead, as figure 9.13 shows, the information about the second element is returned.

Figure 9.13 Running the code from listing 9.15 shows that objects are converted to strings if we try to use them as object properties.

This happens because with objects, keys are stored as strings. This means when we try to use any non-string value, such as an HTML element, as a property of an object, that value is silently converted to a string by calling its `toString` method. Here, this returns the string `"[object HTMLDivElement]"`, and the information about the first element is stored as the value of the `[object HTMLDivElement]` property.

Next, when we try to create a mapping for the second element, a similar thing happens. The second element, which is also an HTML div element, is also converted to a string, and its additional data is also assigned to the `[object HTMLDivElement]` property, overriding the value we set for the first element.

For these two reasons—properties inherited through prototypes and support for string-only keys—plain objects generally aren't useful as maps. Due to this limitation, the ECMAScript committee has specified a completely new type: `Map`.

NOTE Maps are a part of the ES6 standard. For current browser compatibility, see: http://mng.bz/JYYM.

9.2.2 Creating our first map

Creating maps is easy: We use a new, built-in `Map` constructor. Look at the following example.

Listing 9.16 Creating our first map

```
const ninjaIslandMap = new Map();

const ninja1 = { name: "Yoshi"};
const ninja2 = { name: "Hatori"};
const ninja3 = { name: "Kuma"};

ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
ninjaIslandMap.set(ninja2, { homeIsland: "Hokkaido"});

assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
      "The first mapping works");
assert(ninjaIslandMap.get(ninja2).homeIsland === "Hokkaido",
       "The second mapping works");

assert(ninjaIslandMap.get(ninja3) === undefined,
      "There is no mapping for the third ninja!");

assert(ninjaIslandMap.size === 2,
      "We've created two mappings");

assert(ninjaIslandMap.has(ninja1)
    && ninjaIslandMap.has(ninja2),
       "We have mappings for the first two ninjas");
assert(!ninjaIslandMap.has(ninja3),
      "But not for the third ninja!");

ninjaIslandMap.delete(ninja1);
assert(!ninjaIslandMap.has(ninja1)
    && ninjaIslandMap.size() === 1,
     "There's no first ninja mapping anymore!");

ninjaIslandMap.clear();
assert(ninjaIslandMap.size === 0,
       "All mappings have been cleared");
```

Uses the Map constructor to create a map.

Defines three ninja objects.

Creates a mapping for the first two ninja objects by using the map set method.

Gets the mapping for the first two ninja objects by using the map get method.

Checks that there's no mapping for the third ninja.

Checks that the map contains mappings for the first two ninjas, but not for the third one!

Uses the has method to check whether a mapping for a particular key exists.

Uses the delete method to delete a key from the map.

Uses the clear method to completely clear the map.

In this example, we create a new map by calling the built-in `Map` constructor:

```
const ninjaIslandMap = new Map();
```

Next, we create three ninja objects, cleverly called `ninja1`, `ninja2`, and `ninja3`. We then use the built-in map `set` method:

```
ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
```

This creates a mapping between a key—in this case, the `ninja1` object—and a value—in this case, an object carrying the information about the ninja's home island. We do this for the first two ninjas, `ninja1` and `ninja2`.

In the next step, we obtain the mapping for the first two ninjas by using another built-in map method, `get`:

```
assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
      "The first mapping works");
```

The mapping of course exists for the first two ninjas, but it doesn't exist for the third ninja, because we haven't used the third ninja as an argument to the `set` method. The current state of the map is shown in figure 9.14.

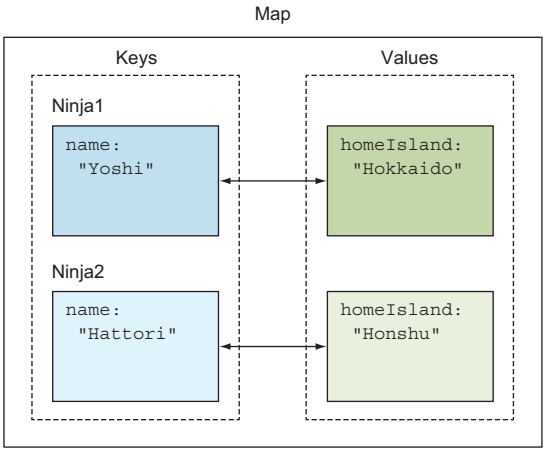


Figure 9.14 A map is a collection of key-value pairs, where a key can be anything—even another object.

In addition to `get` and `set` methods, every map also has a built-in `size` property and `has` and `delete` methods. The `size` property tells us how many mappings we've created. In this case, we've created only two mappings.

The `has` method, on the other hand, notifies us whether a mapping for a particular key already exists:

```
ninjaIslandMap.has(ninja1); //true
ninjaIslandMap.has(ninja3); //false
```

The `delete` method enables us to remove items from our map:

```
ninjaIslandMap.delete(ninja1);
```

One of the fundamental concepts when dealing with maps is determining when two map keys are equal. Let's explore this concept.

KEY EQUALITY

If you come from a bit more traditional background, such as C#, Java, or Python, you may be surprised by the next example.

Listing 9.17 Key equality in maps

```
const map = new Map();
const currentLocation = location.href;

const firstLink = new URL(currentLocation);
const secondLink = new URL(currentLocation);

map.set(firstLink, { description: "firstLink"});
map.set(secondLink, { description: "secondLink"});
```

Uses the built-in location.href property to get the current page URL.

Creates two links to the current page.

Adds a mapping for both links.

```
assert(map.get(firstLink).description === "firstLink",
      "First link mapping" );
assert(map.get(secondLink).description === "secondLink",
       "Second link mapping");
assert(map.size === 2, "There are two mappings");
```

Each link gets its own mapping, even though they point to the same page.

In listing 9.17, we use the built-in `location.href` property to obtain the URL of the current page. Next, by using the built-in URL constructor, we create two new URL objects that link to the current page. We then associate a description object with each link. Finally, we check that the correct mappings have been created, as shown in figure 9.15.

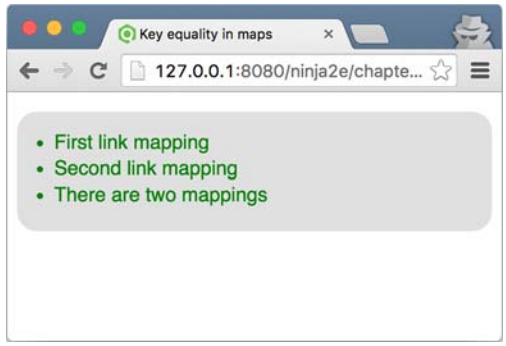


Figure 9.15 If we run the code from listing 9.17, we can see that key equality in maps is based on object equality.

People who have mostly worked in JavaScript may not find this result unexpected: We have two different objects for which we create two different mappings. But notice that the two URL objects, even though they're separate objects, still point to the same URL location: the location of the current page. We could argue that, when creating mappings, these two objects should be considered equal. But in JavaScript, we can't overload the equality operator, and the two objects, even though they have the same content, are always considered different. This isn't the case with other languages, such as Java and C#, so be careful!

9.2.3 Iterating over maps

So far, you've seen some of the advantages of maps: You can be sure they contain only items that you put in them, and you can use anything as a key. But there's more!

Because maps are collections, there's nothing stopping us from iterating over them with `for...of` loops. (Remember, we used the `for...of` loop to iterate over values created by generators in chapter 6.) You're also guaranteed that these values will be visited in the order in which they were inserted (something we can't rely on when iterating over objects using the `for...in` loop). Let's look at the following example.

Listing 9.18 Iterating over maps

Creates a new map, just as we've done so far.

```
const directory = new Map();

directory.set("Yoshi", "+81 26 6462");
directory.set("Kuma", "+81 52 2378 6462");
directory.set("Hiro", "+81 76 277 46");
```

Creates a ninja directory that stores each ninja's phone number.

```
for(let item of directory){
  assert(item[0] !== null, "Key:" + item[0]);
  assert(item[1] !== null, "Value:" + item[1]);
}
```

Iterates over each item in a dictionary using the for...of loop. Each item is a two-item array: a key and a value.

```
for(let key of directory.keys()){
  assert(key !== null, "Key:" + key);
  assert(directory.get(key) != null,
        "Value:" + directory.get(key));
}
```

We can also iterate over keys using the built-in keys method…

```
for(var value of directory.values()){
  assert(value !== null, "Value:" + value);
}
```

…and over values using the built-in values method.

As the previous listing shows, once we've created a mapping, we can easily iterate over it using the `for...of` loop:

```
for(var item of directory){
          assert(item[0] !== null, "Key:" + item[0]);
          assert(item[1] !== null, "Value:" + item[1]);
}
```

In each iteration, this gives a two-item array, where the first item is a key and the second item is the value of an item from our directory map. We can also use the `keys` and `values` methods to iterate over, well, keys and values contained in a map.

Now that we've looked at maps, let's visit another newcomer to JavaScript: *sets*, which are collections of unique items.

9.3 Sets

In many real-world problems, we have to deal with collections of *distinct* items (meaning each item can't appear more than once) called *sets*. Up to ES6, this was something you had to implement yourself by mimicking sets with standard objects. For a crude example, see the next listing.

Listing 9.19 Mimicking sets with objects

```
function Set(){
  this.data = {};
  this.length = 0;
}
```

Uses an object to store items

```
Set.prototype.has = function(item){
  return typeof this.data[item] !== "undefined";
};
```

Checks whether the item is already stored

```
Set.prototype.add = function(item){
  if(!this.has(item)){
    this.data[item] = true;
    this.length++;
  }
};
```

Adds an item only if it isn't already contained in the set

```
Set.prototype.remove = function(item){
  if(this.has(item)){
    delete this.data[item];
    this.length--;
  }
};

const ninjas = new Set();
ninjas.add("Hattori");
ninjas.add("Hattori");

assert(ninjas.has("Hattori") && ninjas.length === 1,
       "Our set contains only one Hattori");

ninjas.remove("Hattori");
assert(!ninjas.has("Hattori") && ninjas.length === 0,
       "Our set is now empty");
```

Removes an item it it's already contained in the set

Tries to add Hattori twice

Checks that Hattori was added only once

Removes Hattori and checks that he was removed from the set

Listing 9.19 shows a simple example of how sets can be mimicked with objects. We use a data-storage object, `data`, to keep track of our set items, and we expose three methods: `has`, which checks whether an item is already contained in the set; `add`, which adds an item only if the same item isn't already contained in the set; and `remove`, which removes an already-contained item from the set.

But this is a poor doppelganger. Because with maps, you can't really store objects—only strings and numbers—and there's always the risk of accessing prototype objects. For these reasons, the ECMAScript committee decided to introduce a completely new type of collection: *sets*.

NOTE Sets are a part of the ES6 standard. For current browser compatibility, see http://mng.bz/QRTS.

9.3.1 Creating our first set

The cornerstone of creating sets is the newly introduced constructor function, conveniently named `Set`. Let's look at an example.

Listing 9.20 Creating a set

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);

assert(ninjas.has("Hattori"), "Hattori is in our set");
assert(ninjas.size === 3, "There are only three ninjas in our set!");
```

The Set constructor can take an array of items with which the set is initialized.

Discards any duplicate items.

```
assert(!ninjas.has("Yoshi"), "Yoshi is not in, yet..");
ninjas.add("Yoshi");
assert(ninjas.has("Yoshi"), "Yoshi is added");
assert(ninjas.size === 4, "There are four ninjas in our set!");

assert(ninjas.has("Kuma"), "Kuma is already added");
ninjas.add("Kuma");
assert(ninjas.size === 4, "Adding Kuma again has no effect");

for(let ninja of ninjas) {
  assert(ninja !== null, ninja);
}
```

We can add new items that aren't already contained in the set.

Adding existing items has no effect.

Iterates through the set with a for...of loop.

Here we use the built-in `Set` constructor to create a new `ninjas` set that will contain distinct ninjas. If we don't pass in any arguments, an empty set is created. We can also pass in an array, such as this, which pre-fills the set:

```
new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

As we already mentioned, sets are collections of unique items, and their primary purpose is to stop us from storing multiple occurrences of the same object. In this case, this means `"Hattori"`, which we tried to add twice, is added only once.

A number of methods are accessible from every set. For example, the `has` method checks whether an item is contained in the set:

```
ninjas.has("Hattori")
```

and the `add` method is used to add unique items to the set:

```
ninjas.add("Yoshi");
```

If you're curious about how many items are in a set, you can always use the `size` property.

Similar to maps and arrays, sets are collections, so we can iterate over them with a `for...of` loop. As you can see in figure 9.16, the items are always iterated over in the order in which they were inserted.

Now that we've gone through the basics of sets, let's visit some common operations on sets: unions, intersections, and differences.

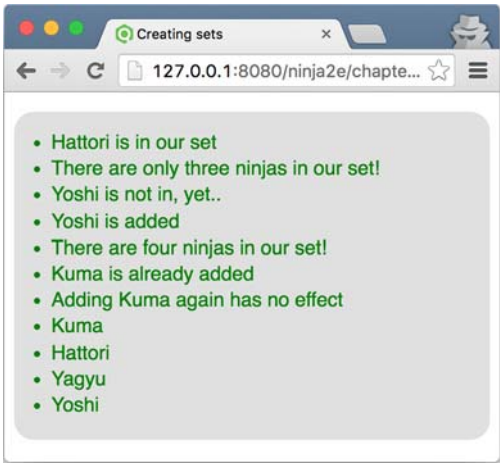


Figure 9.16 Running the code from listing 9.20 shows that the items in a set are iterated over in the order in which they were inserted.

9.3.2 Union of sets

A union of two sets, `A` and `B`, creates a new set that contains all elements from both `A` and `B`. Naturally, each item can't occur more than once in the new set.

Listing 9.21 Using sets to perform a union of collections

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = ["Hattori", "Oda", "Tomoe"];

const warriors = new Set([...ninjas, ...samurai]);

assert(warriors.has("Kuma"), "Kuma is here");
assert(warriors.has("Hattori"), "And Hattori");
assert(warriors.has("Yagyu"), "And Yagyu");
assert(warriors.has("Oda"), "And Oda");
assert(warriors.has("Tomoe"), "Tomoe, last but not least");

assert(warriors.size === 5, "There are 5 warriors in total");
```

Creates an array of ninjas and samurai. Notice that Hattori is both a ninja and a samurai.

Creates a new set of warriors by spreading ninjas and samurai.

All the ninjas and samurai are included in the new warriors set.

There are no duplicates in the new set. Even though Hattori is in both the ninjas and samurai sets, he is included only once.

We first create an array of `ninjas` and an array of `samurai`. Notice that Hattori is leading a busy life: samurai by day, ninja by night. Now imagine that we need to create a collection of people whom we can call to arms if a neighboring daimyo decides that his province is a bit cramped. We create a new set, `warriors`, that includes all ninjas and all samurai. Hattori is in both collections, but we want to include him only once—it's not like two Hattoris will respond to our call.

In this case, a set is perfect! We don't need to manually keep track of whether an item has been already included: The set takes care of that by itself, automatically. When creating this new set, we use the spread operator `[...ninjas, ...samurai]` (remember chapter 3) to create a new array that contains all ninjas and all samurai. In case you're wondering, Hattori is present twice in this new array. But when we finally pass that array to the `Set` constructor, Hattori is included only once, as shown in figure 9.17.

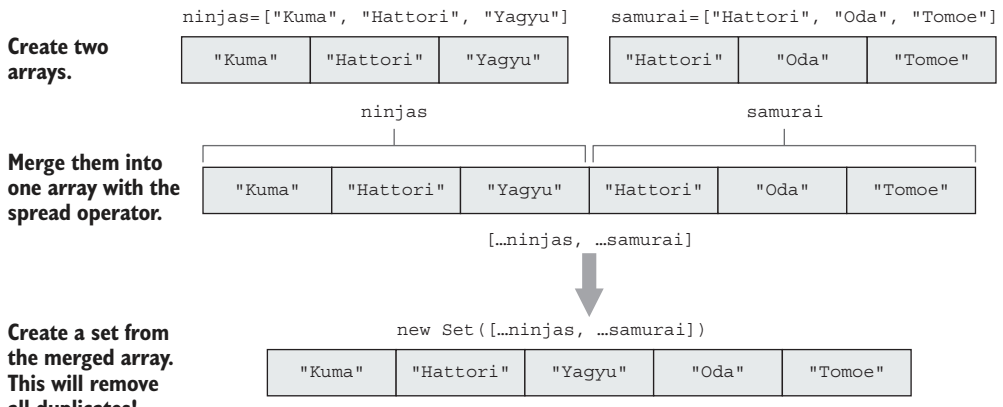


Figure 9.12 A union of two sets keeps the items from both collections (without duplicates, of course).

9.3.3 Intersection of sets

The *intersection* of two sets, `A` and `B`, creates a set that contains elements of `A` that are also in `B`. For example, we can find ninjas that are also samurai, as shown next.

Listing 9.22 Intersection of sets

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const ninjaSamurais = new Set(
  [...ninjas].filter(ninja => samurai.has(ninja))
);

assert(ninjaSamurais.size === 1, "There's only one ninja samurai");
assert(ninjaSamurais.has("Hattori"), "Hattori is his name");
```

Uses the spread operator to turn our set into an array so we can use the array's filter method to keep only ninjas that are contained in the samurai set

The idea behind listing 9.22 is to create a new set that contains only ninjas who are also samurai. We do this by taking advantage of the array's `filter` method, which, as you'll remember, creates a new array that contains only the items that match a certain criterion. In this case, the criterion is that the ninja is also a samurai (is contained in the set of samurai). Because the `filter` method can only be used on arrays, we have to turn the `ninjas` set into an array by using the spread operator:

```
[...ninjas]
```

Finally, we check that we've found only one ninja who's also a samurai: the Jack of all trades, Hattori.

9.3.4 Difference of sets

The difference of two sets, `A` and `B`, contains all elements that are in set `A` but are *not* in set `B`. As you might guess, this is similar to the intersection of sets, with one small but significant difference. In the next listing, we want to find only true ninjas (not those who also moonlight as samurai).

Listing 9.23 Difference of sets

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const pureNinjas = new Set(
  [...ninjas].filter(ninja => !samurai.has(ninja))
);

assert(pureNinjas.size === 2, "There's only one ninja samurai");
assert(pureNinjas.has("Kuma"), "Kuma is a true ninja");
assert(pureNinjas.has("Yagyu"), "Yagyu is a true ninja");
```

With set difference, we care only about ninjas who are not samurai!

The only change is to specify that we care only about the ninjas who are *not* also samurai, by putting an exclamation mark (`!`) before the `samurai.has(ninja)` expression.

9.4 Summary

- Arrays are a special type of object with a `length` property and `Array.prototype` as their prototype.
- We can create new arrays using the array literal notation (`[]`) or by calling the built-in `Array` constructor.
- We can modify the contents of an array using several methods accessible from array objects:
  - The built-in `push` and `pop` methods add items to and remove items from the end of the array.
  - The built-in `shift` and `unshift` methods add items to and remove items from the beginning of the array.
  - The built-in `splice` method can be used to remove items from and add items to arbitrary array positions.
- All arrays have access to a number of useful methods:
  - The `map` method creates a new array with the results of calling a callback on every element.
  - The `every` and `some` methods determine whether all or some array items satisfy a certain criterion.
  - The `find` and `filter` methods find array items that satisfy a certain condition.
  - The `sort` method sorts an array.
  - The `reduce` method aggregates all items in an array into a single value.
- You can reuse the built-in array methods when implementing your own objects by explicitly setting the method call context with the `call` or `apply` method.
- Maps and dictionaries are objects that contain mappings between a key and a value.
- Objects in JavaScript are lousy maps because you can only use string values as keys and because there's always the risk of accessing prototype properties. Instead, use the new built-in `Map` collection.
- Maps are collections and can be iterated over using the `for...of` loop.
- Sets are collections of unique items.

9.5 Exercises

1. What will be the content of the samurai array, after running the following code?

```
const samurai = ["Oda", "Tomoe"];
samurai[3] = "Hattori";
```

2. What will be the content of the ninjas array, after running the following code?

```
const ninjas = [];

ninjas.push("Yoshi");
```

```
ninjas.unshift("Hattori");

ninjas.length = 3;

ninjas.pop();
```

3 What will be the content of the samurai array, after running the following code?

```
const samurai = [];

samurai.push("Oda");
samurai.unshift("Tomoe");
samurai.splice(1, 0, "Hattori", "Takeda");
samurai.pop();
```

4 What will be stored in variables first, second, and third, after running the following code?

```
const ninjas = [{name:"Yoshi", age: 18},
    {name:"Hattori", age: 19},
    {name:"Yagyu", age: 20}];

const first = persons.map(ninja => ninja.age);
const second = first.filter(age => age % 2 == 0);
const third = first.reduce((aggregate, item) =>  aggregate + item, 0);
```

5 What will be stored in variables first and second, after running the following code?

```
const ninjas = [{ name: "Yoshi", age: 18 },
                { name: "Hattor", age: 19 },
                { name: "Yagyu", age: 20 }];

const first = ninjas.some(ninja => ninja.age % 2 == 0);
const second = ninjas.every(ninja => ninja.age % 2 == 0);
```

6 Which of the following assertions will pass?

```
const samuraiClanMap = new Map();

const samurai1 = { name: "Toyotomi"};
const samurai2 = { name: "Takeda"};
const samurai3 = { name: "Akiyama"};

const oda = { clan: "Oda"};
const tokugawa = { clan: "Tokugawa"};
const takeda ={clan: "Takeda"};

samuraiClanMap.set(samurai1, oda);
samuraiClanMap.set(samurai2, tokugawa);
samuraiClanMap.set(samurai2, takeda);
```

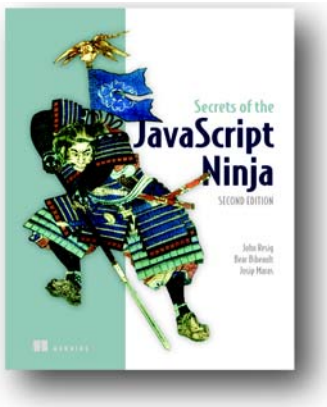
```
assert(samuraiClanMap.size === 3, "There are three mappings");
assert(samuraiClanMap.has(samurai1), "The first samurai has a mapping");
assert(samuraiClanMap.has(samurai3), "The third samurai has a mapping");
```

7 Which of the following assertions will pass?

```
const samurai = new Set("Toyotomi", "Takeda", "Akiyama", "Akiyama");
assert(samurai.size === 4, "There are four samurai in the set");

samurai.add("Akiyama");
assert(samurai.size === 5, "There are five samurai in the set");

assert(samurai.has("Toyotomi", "Toyotomi is in!");
assert(samurai.has("Hattori", "Hattori is in!");
```



More than ever, the web is a universal platform for all types of applications, and JavaScript is the language of the web. If you're serious about web development, it's not enough to be a decent JavaScript coder. You need to be ninja-stealthy, efficient, and ready for anything.

Secrets of the JavaScript Ninja, Second Edition takes you below the surface and helps you understand the deceptively complex world of JavaScript and browser-based application development. You'll skip the basics and dive into core JavaScript concepts such as functions, closures, objects, prototypes, promises, and so on. With examples, illustrations, and insightful explanations, you'll benefit from the collective wisdom of seasoned experts John Resig, Bear Bibeault, and Josip Maras.

What's inside

- Discover how the individual parts of a web application come together on the browser as a platform
- Learn how functions, objects, and closures work in JavaScript and how you can use them to write simpler, more effective code
- Anticipate common application pitfalls and discover how to avoid them
- Write succinct code for text processing with regular expressions
- Manage asynchronous code with promises
- Use arrays efficiently, with a focus on succinct functionally-oriented API methods
- Embrace the concepts from ES6
- Grok the browser infrastructure for events, timers, web workers, and the DOM so you can write more performant applications
- Understand the difficulty of cross-browser development and learn techniques for developing cross-browser strategies

This book is written for readers familiar with JavaScript who want to take the next step toward ninjahood.

Getting started with Gulp

To maximize your overall productivity and efficiency, you also need to learn the concepts and tools associated with task automation and with creating a client-side build process. The following chapter provides a comprehensive introduction to Gulp.js, a Node.js-based task runner that can automate and simplify even the most complex workflows.

Chapter 2 from *Front-End Tooling with Gulp, Bower, and Yeoman*
by Stefan Baumgartner

Getting started with Gulp

This chapter covers

- An introduction to Gulp
- The concepts of streams
- Creating simple tasks to automate tools
- Creating execution pipelines for multifunctional tasks

You start your new workflow by concentrating on the build system. Although this was the last step in the previous chapter, it serves as the necessary foundation for the upcoming elements of our new workflow. The build system deals with a good chunk of tasks that occur in our day-to-day workflow, like concatenating, minifying, and testing our code. The build tool is initialized by the scaffolding tool and integrates the dependency manager's components into the project. Figure 2.1 shows the part of the build tool in our tooling workflow again.

Gulp is the foundation for your workflow automation. It's the first element that's being initialized by the scaffolding tool, and it integrates components into the project. Also, because it's creating an actively used development environment, it will become one of the most used and integral parts of your development cycles.

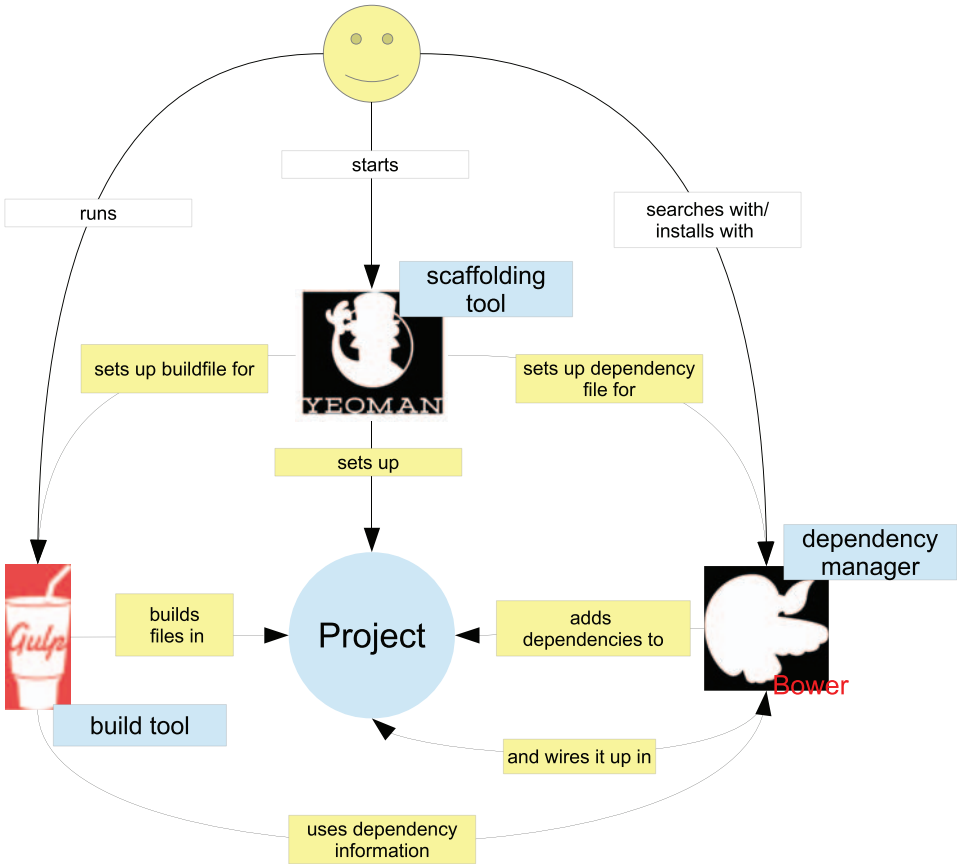


Figure 2.1 Your workflow setup from chapter 1

Gulp is a build tool written in JavaScript and running on Node.js. Because Gulp is a JavaScript program, and Gulp's build instructions are also written in JavaScript, it's very close to the environment JavaScript developers use every day. This makes Gulp a perfect choice for automating the daily tasks of a front end developer. In this chapter, I'll show you how to set up Gulp for your projects. You'll learn about the individual bits and pieces a Gulp setup consists of, and you'll develop your own building instructions for your JavaScript applications and CSS files.

2.1 Setting up Gulp

Gulp as a Node.js tool is a conglomeration of little pieces that make up a whole. In this section you'll learn about the different building blocks of Gulp and how to install both the command-line interface and the local Gulp installation.

Sample project and Node.js

For all the upcoming samples in this book, I've prepared a sample project, which you can find under http://github.com/frontend-tooling/sample-project-gulp. If you're familiar with Git, you can use the `clone` command to get its contents. Otherwise, there's a Download Zip button on the website you can use to get the files onto your system.

To re-create the samples, you need to have Node.js installed and ready on your system. You'll find the necessary binaries and installation instructions on https://www.nodejs.org for your system (Windows, Linux, or Mac OSX).

2.1.1 The building blocks of Gulp

Build tools in general consist of at least two parts: the tool executing the build and a build file containing all the instructions for the build tool. Gulp is no exception: the tool executing the build is called Gulp. Gulp's build file is commonly referred to as a Gulpfile. Gulp itself can be broken down in several other parts. Figure 2.2 shows a quick overview of Gulp's parts and how they interact.

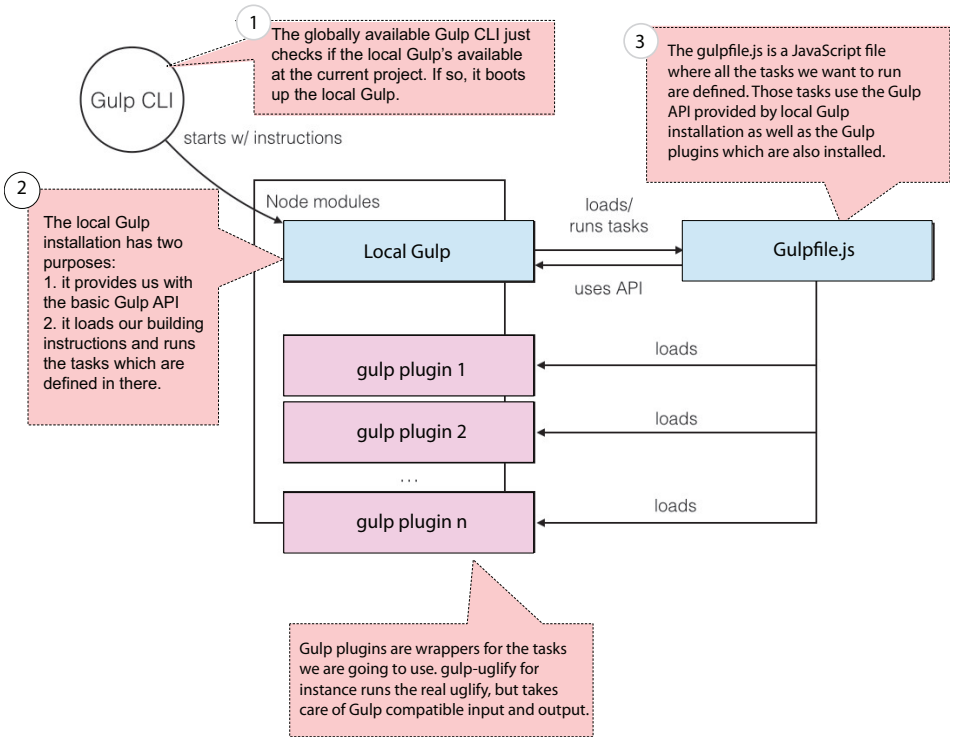


Figure 2.2 The Gulp CLI starts the local Gulp installation with the parameters provided on the command line. The local installation takes the local Gulpfile, which in turn loads Gulp plugins and defines tasks using the Gulp API. Gulp itself runs and loads these tasks.

So a complete Gulp installation consists of the following parts:

- *The Gulp CLI*—A command-line interface to start the build tool
- *The local Gulp installation*—The actual software running your build
- *The Gulpfile*—The building instructions that tell you how to build your software
- *Gulp plugins*—Lots and lots of tiny executables that know how to combine, modify, and assemble parts of your software

That's quite a lot! But don't worry; in this chapter we'll deal with all of these. Let's start with the CLI.

2.1.2 The Gulp command-line interface

Node.js modules can be installed globally. That makes them usable as a command-line executable tool. Or you can install modules locally as a library for your own projects. Both ways are shown in figure 2.3.

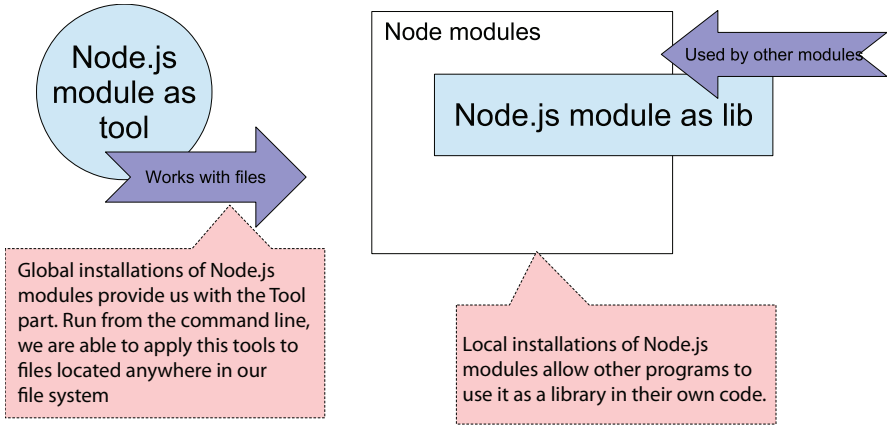


Figure 2.3 The two ways a Node.js module can be installed. If you install a module globally, it just provides its functionality as an executable from the command line, working with the files you want to change. It becomes a tool. Installed locally, you can use the same functionality but in your own programs.

The same goes for Gulp, with the exception that Gulp has two separate packages for the global tool part and the library. One is the Gulp command-line interface; it's installed globally and can be executed from your terminal/bash/command prompt. The other package is the local Gulp installation, which is the actual Gulp runtime, handling all the tasks and providing the entire API. Gulp's command-line interface provides a *global* entry point for the *local* Gulp installation, forwarding all the parameters entered to the local installation and kicking off the version of the runtime you've installed for your project. From there on, the local installation takes the lead and executes your build.

Gulp's CLI is rather dumb, meaning that its only functionality is to check for a local installation that it can execute. The reason behind this is to allow for a version-independent execution. Image you have a project running the legacy version of Gulp 3.8. The CLI will be able to execute this project because the interface to the local Gulp installation is the same. A newer project running on Gulp 4 can be executed with the same CLI. In the end, you'll most likely never update your command-line interface but will be able to run all Gulp projects you'll ever create, no matter which local installation they require.

To install it, boot up your bash, terminal, or Windows command prompt (it works on all of them) and check to see if Node.js is installed:

```
$ node --version
```

If you have Node.js installed, this command should output the current version of your installation. Next, you need to check for Node's package manager, NPM. It comes with Node.js; check for it with this command:

```
$ npm --version
```

Again, you should get a correct version output. Should one of those programs not be available, please check appendix B for installation instructions or visit the Node.js website at https://nodejs.org. With Node.js and NPM installed and your command line booted up, install the Gulp CLI with

```
$ npm install -g gulp-cli
```

Note the `-g` parameter after `install`. It tells your Node package manager to make this installation globally available. Once NPM has finished, you have a new command available on your system. Type the following on your command line to make sure the installation worked:

```
$ gulp --version
```

It should output something like this:

```
[12:04:15] CLI version 0.2.0
```

The first step is done. You have the Gulp CLI installed on your system! Let's continue with the local installation.

2.1.3 The local Gulp installation

The local Gulp installation has two main purposes: loading and executing the building instructions written in the Gulpfile and exposing an API that can be used by the Gulpfile. The local Gulp installation is the actual software executing your builds. The global installation just kicks off the local software installed separately for each project. Figure 2.4 illustrates this.

To install Gulp locally, open your command line and move to the directory where you unzipped (or cloned) our sample project—not in the app folder directly but one level up where the README.md file is located. There, promote the whole folder to a Node module by typing

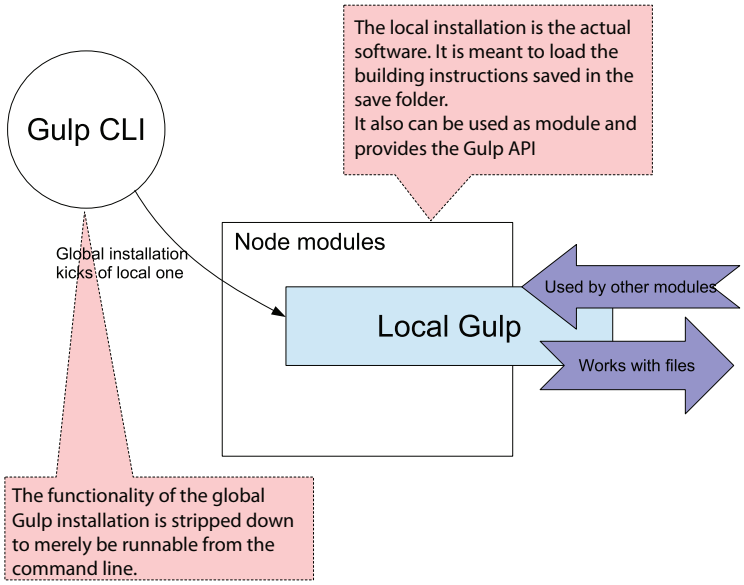


Figure 2.4 The global installation of Gulp is the Gulp CLI. Its purpose is to check the availability of a local installation, which it starts on call. The local Gulp is located in the local node_modules folder of a JavaScript project. It contains all the necessary runtime functions and provides an API for build files (Gulpfiles).

```
$ npm init
```

What follows is a short questionnaire asking you several pieces of information about your project, but because you probably don't want to publish your new module to the NPM registry (at least not now), you can leave everything at its default value. Once you've finished, you'll see that a new file called package.json is available in your folder. This file will store all the information on which Node modules are necessary for your application—the so-called dependencies—and which version they have to be. This file is the core of every Node project, and some plugins access it directly to get information on installed modules.

The package.json file stores the information of the Node modules your project depends on. It divides this information into runtime dependencies (modules the project needs to work properly) and development dependencies (modules you need to develop your project). Because your build tool falls into the latter category, you install the local Gulp installation with the following command:

```
$ npm install --save-dev gulp
```

Gulp is downloaded, and the `save-dev` parameter stores the correct version in your package.json file:

```
{
  "name": "sample-project-gulp",
```

```
  "version": "1.0.0",
  "description": "The sample project we will use throughout the Gulp chapters",
  ...
  "devDependencies": {
    "gulp": "^4.0.0"
  }
}
```

Doing a version check again (with `gulp --version`), you can see that the output has changed. The Gulp CLI recognizes your local installation:

```
[20:40:12] CLI version 0.2.0
[20:40:12] Local version 4.0.0
```

> **Gulp 4**
>
> This chapter and some parts of the book were written with the newest version of Gulp, version 4, in mind. But at the time this book went into production, Gulp 4 was still in a pre-release state. So we don't know if at the time of this book's release Gulp 4 will have been released to the public. If `gulp --version` gives you a 3.x version number, then Gulp 4 is still in the pre-release state. To install Gulp 4, use this command instead:
>
> ```
> $ npm install --save-dev gulpjs/gulp.git#4.0
> ```
>
> This will download and install the pre-release branch.

You now have both the CLI and the local Gulp installed. The next step is working on your Gulpfile.

2.2 Creating Gulpfiles

In the previous section you set up the basic software that runs Gulp. You installed the global CLI and made sure that the local Gulp installation is available. The chain up until now is to start the local Gulp installation with the globally available CLI (figure 2.4). Now you're ready for the next step in the chain: the Gulpfile.

2.2.1 A Gulp "Hello World" task

The Gulpfile is a JavaScript file containing all your building instructions. Building instructions contain a series of commands that can be bundled into tasks. In Gulp, a *task* is a plain JavaScript function containing all the commands you want to execute. Any function will do, as long as it's defined using the first method of Gulp's API: the `gulp.task` method. Figure 2.5 shows the method's signature.

The `gulp.task` method serves one purpose: it gives the task function—which is just plain JavaScript—a unique name. This name pushes the function into Gulp's execution space. In doing so, Gulp "knows" of this function's existence and can use this

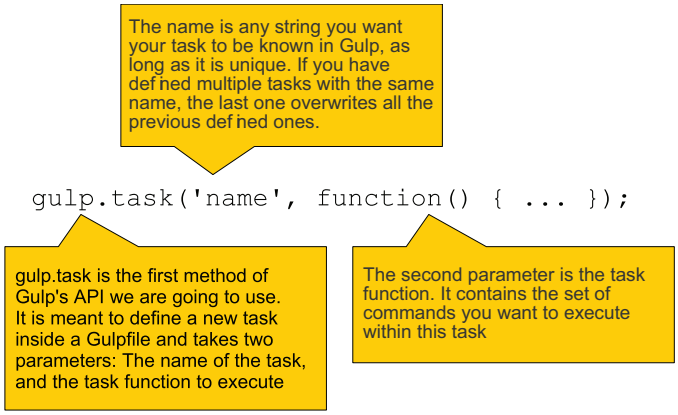


Figure 2.5 The signature of the `gulp.task` method

reference to execute it. This means that `gulp.task` provides a direct interface from the command line to that function.

Let's write your first task. In true programming tradition, you'll go for the output of "Hello World!" on the command line. Create a new, empty text file called Gulpfile.js in the same folder where your package.json and node_modules folders are located. The following listing shows the contents to add.

Listing 2.1 Listing 2.1 Hello World in Gulp–Gulpfile.js

```
var gulp = require('gulp');

gulp.task('test', function () {
  console.log('Hello World!');
});
```

❶ Require the local Gulp installation in your Gulpfile

❷ Define a new task named test.

Print "Hello World!" on the command line

❶ In requiring the local Gulp installation in your Gulpfile, you have Gulp's API available. ❷ Gulp and the Gulpfile are here inherently connected: Gulp loads the Gulpfile to know which tasks are available, and the Gulpfile loads Gulp to use its API. In defining the `test` task, the task is available within Gulp. The second parameter is the function you want to execute.

Gulp and the Gulpfile share a unique connection: whereas Gulp needs the Gulpfile to know which tasks are available and can be executed, the Gulpfile needs Gulp to have access to the API. See figure 2.6 for details.

With `gulp.task`, the first method provided by Gulp's API, you can promote task functions to Gulp tasks. In listing 2.1 you created the "Hello World" task that now runs by the name `test`. This name is available in Gulp's execution space, and you can refer to it directly when calling Gulp from the command line. With the command

```
$ gulp test
```

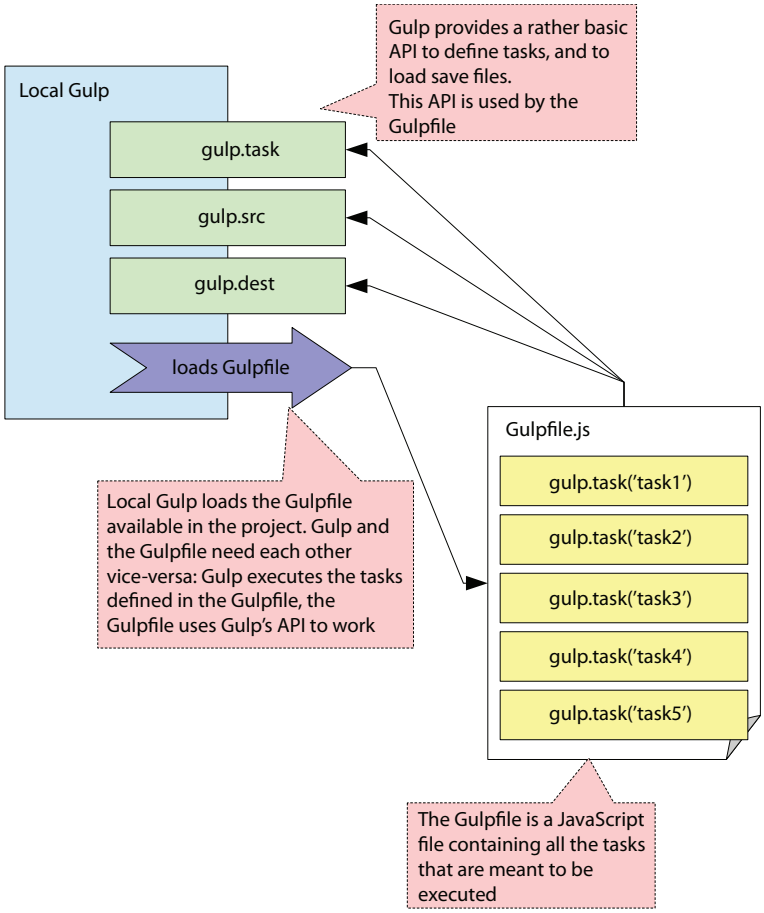


Figure 2.6 The local Gulp installation interplays with the Gulpfile. It loads the Gulpfile available in the project and executes the available tasks, as instructed by the command-line interface. It also provides the Gulpfile with a basic API, which is needed to create and define tasks.

you are able to execute this particular task. The various parts of executing tasks kick in (as shown in figure 2.7) as follows:

1. The global Gulp CLI loads the local Gulp installation.
2. The local Gulp installation loads the Gulpfile.
3. The Gulpfile loads the local Gulp installation and defines a new task called `test`.
4. The local Gulp installation is passed a command-line parameter. This is the name of the task to execute.
5. Because the task of the same name is available, the local Gulp executes the function attached to it.

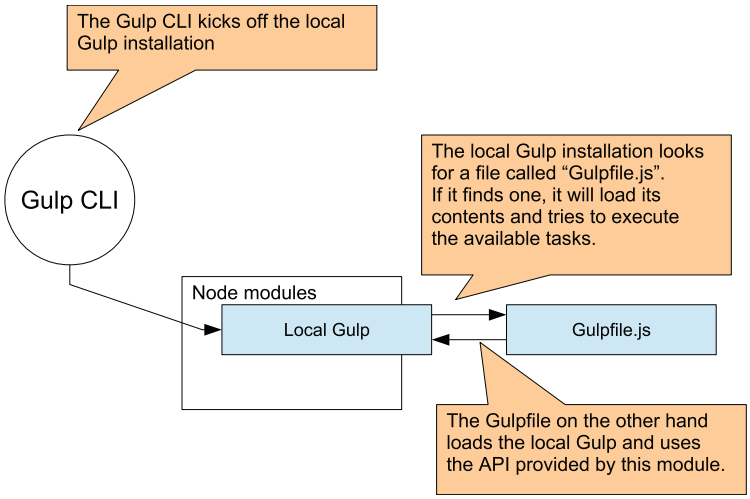


Figure 2.7 The global CLI kicks off the local Gulp. The local Gulp looks for a Gulpfile and loads its contents.

The output looks like this:

```
[15:01:06] Using gulpfile ~/Project/playground/test/gulpfile.js
[15:01:06] Starting 'test'...
Hello World
[15:01:06] Finished 'test' after 95 µs
```

You can now execute functions defined in your Gulpfile via the command line. Of course, a "Hello World!" isn't something you need a build tool for. Let's do something useful with it.

2.2.2 Dealing with streams

With Gulp, you want to read input files and transform them into the desired output, loading lots of JavaScript files and combining them into one. The Gulp API provides some methods for reading, transforming, and writing files, all using streams under the hood.

Streams are a fairly old concept in computing, originating from the early Unix days in the 1960s. A *stream* is a sequence of data coming over time from a source and running to a destination. The source can be of multiple types: files, the computer's memory, or input devices like a keyboard or a mouse. Once a stream is opened, data flows in chunks from its origin to the process consuming it. Coming from a file, every character or byte would be read one at a time; coming from the keyboard, every keystroke would transmit data over the stream. The biggest advantage compared to loading all the data at once is that, in theory, the input can be endless and without limits. Coming from a keyboard that makes total sense: why should anybody close the input

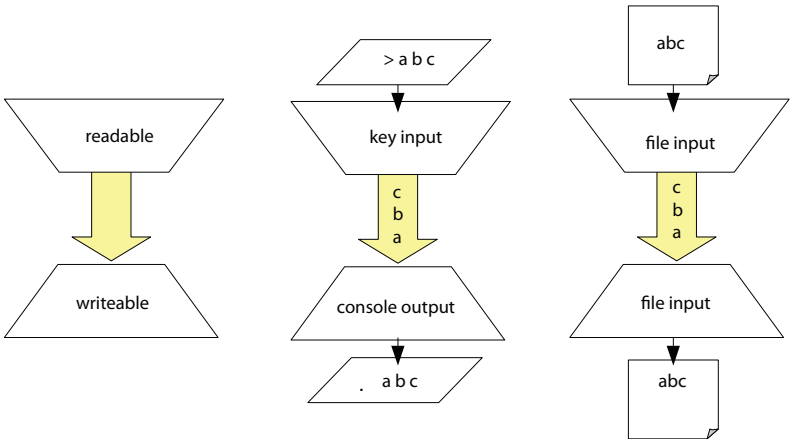


Figure 2.8 Streams can be of two types: readable streams, which access data, and writeable streams, which write data.

stream you're using to control your computer? Input streams are also called readable streams, indicating that they're meant to read data from a source. On the other hand, there are outbound streams or destinations: they can also be files, some place in memory, but also output devices like the command line, a printer, or your screen. They're also called writeable streams, meaning that they're meant to store the data that comes over the stream. Figure 2.8 illustrates how streams work.

The data is a sequence of elements made available over time (like characters or bytes). Readable streams can originate from different sources, such as input devices (keyboards), files, or data stored in memory. Writeable streams can also end in different places, such as files and memory again, but also the command line. Readable and writeable streams can be interchanged: keyboard input can end up in a file, and file input can end up on the command line.

Not only is it possible to have an endless amount of input, but you also can combine different readable and writeable streams. Key input can be directly stored into a file, or you can print file input out to the command line or even a connected printer. The interface stays the same no matter what the sources or destinations are.

2.2.3 Readable and writeable streams with Gulp

In Gulp, you can use the `gulp.src` method to create readable file streams. It allows you to select which files you want to process in your task. Because you're going to deal with many different files and most likely won't know how your files are called directly, you can define some selection patterns using *globs*. The counterpart for `gulp.src` and your writeable stream is `gulp.dest`. Here you define where you want to put your files. The parameter `gulp.dest` takes is a simple string pointing to the directory, relative to the directory where the Gulpfile is located. Should this directory not be available, Gulp will create it accordingly. Figure 2.9 shows this process.

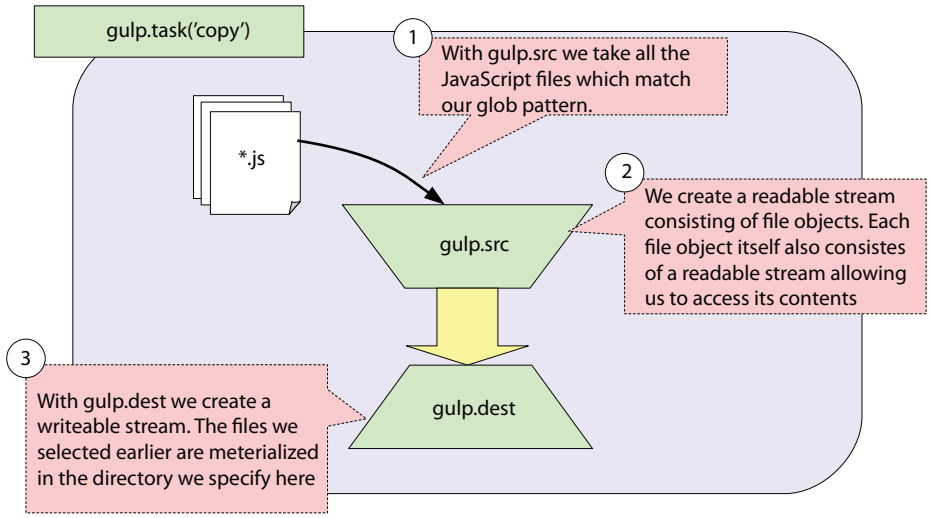


Figure 2.9 A basic Gulp task working with `gulp.src` and `gulp.dest`. You read files from one place and pipe their contents to a destination.

Globs

Globs are a well-known concept in computer science and programming, and if you've ever deleted all files of a certain type on your command line, then you're familiar with how they work. The asterisk in `*.js` stands for "everything" and is a wildcard. With that, every JavaScript file in the app/scripts folder gets selected. Node globs are more advanced, though, and allow for more sophisticated patterns. One such pattern is the double asterisk right before the `*.js` part. This pattern, called *globstar*, is also a wildcard that tells your selection engine to select practically everything, but in this case you want to match for zero or more directories in that particular subdirectory. For example, the glob `app/scripts/*.js` with a single wildcard before the filename allows you to select any JavaScript file inside the scripts directory but not its subfolders. Using the globstar pattern `app/scripts/**/*.js`, you include JavaScript files in subdirectories.

So far you know that `gulp.src` reads files according to a pattern and `gulp.dest` stores files in a certain directory. With this knowledge you can create your first Gulp task that makes use of this part of Gulp's API. `gulp.src` reads files and `gulp.dest` writes files, so a combination of both copies files from one point to the other. See the next listing for a sample implementation.

Listing 2.2 Copy files from one directory to the other—Gulpfile.js

```
var gulp = require('gulp');

gulp.task('copy', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(gulp.dest('dist'));
});
```

❶ Create a new task called copy
❷ Create a new readable stream of file objects
❸ Streams provide a pipe function

❶ In creating the `copy` task, the following function is available in Gulp's task execution space. ❷ The glob pattern provided selects all files ending with "js" in all subdirectories of app/scripts (including files inside app/scripts). ❸ You can pipe the contents of a stream through to other functions. In this case, you pipe them through Gulp's `gulp.dest` function. This function materializes all files inside your stream in the specified directory. In this case, you save them in the dist folder. If the folder isn't available, it will be created.

> **NOTE** Running `gulp copy` from the command line kicks off your execution chain again, this time running the `copy` task.

`gulp.src` opens a readable stream of files. The chunks of data processed are all the files selected by the glob specified in the first parameter. Once this stream is opened, you can steer your stream toward a certain process using a pipe. Pipes are not a Gulp specialty per se but rather a concept used by Node streams in general. In listing 2.2 you piped the contents to the `gulp.dest` function. The `gulp.dest` function opens a writeable stream. Writeable streams are meant as a sink for data, the end point for the data to stay. This can be output on the screen but also in a file. In this case, it's output on the file system. The writeable stream created from `gulp.dest` accepts the same type of data that the readable stream from `gulp.src` creates.

So you read files from the file system and stored them back into the file system, creating a copy functionality. You'd agree that copying is essential but in the end boring. Let's spice it up with transformable streams in the next section.

2.3 Handling tasks with Gulp plugins

So far you've used Gulp as a layer for running functions from the command line and reading files from one place on the file system and writing them back to another place. But Gulp's true power comes when you start to use Gulp plugins. Gulp plugins are little pieces of software that allow you to transform the files in a stream. This section shows you the possibilities and technologies used by Gulp.

2.3.1 Transforming data

Streams are not just good for transferring data between different input sources and output destinations. With the data exposed once a stream is opened, developers can transform the data that comes from the stream before it reaches its destination, such as transforming all lowercase characters in a file to uppercase characters.

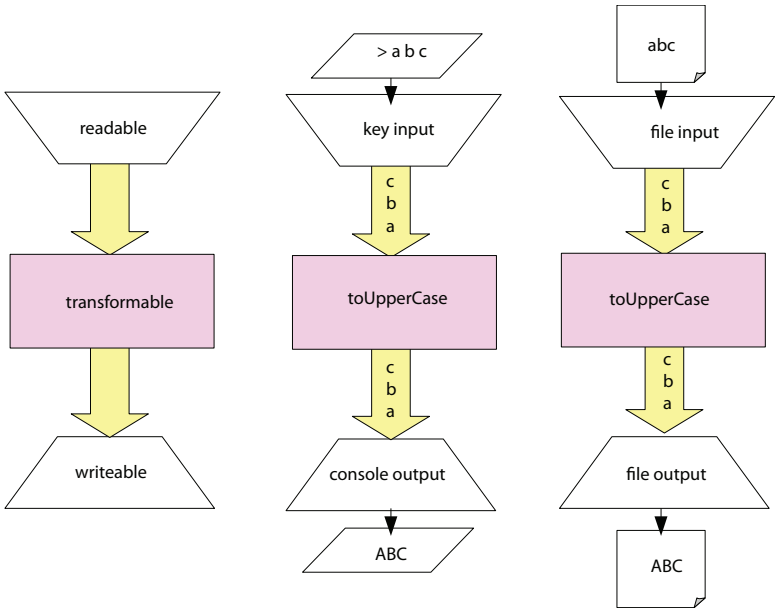


Figure 2.10 Streams are good not only for transferring data but also for modifying it.

This is one of the greatest powers of streams. Once a stream is opened and you can read the data piece by piece, you can slot different programs in between. Figure 2.10 illustrates this process.

To modify data you add transformation blocks between the input and the output. In this example you get your input data from different sources and channel it through a `toUpperCase` transformation. This changes lowercase characters to their uppercase equivalent. Those blocks can be defined once and reused for different input origins and outputs.

In Gulp, transformation is done via plugins. The Gulp ecosystem contains more than 1500 plugins that allow you to transform data in various ways. Let's see how you can transform your JavaScript files using a common transformation in the JavaScript world: Uglify.

Uglify is a minification library written in JavaScript. It removes all unnecessary white spaces, reduces variables and function names to a possible minimum while keeping global APIs intact, and takes on every JavaScript code optimization in the book (like writing `true` as `!0`, for example, because it saves 2 bytes!). Your code gets reduced to an absolute minimum, which allows for faster parsing and transfer over the network. For example, the popular jQuery library gets pared down from roughly 250 KB to 90 KB, which is a little more than a third of its original size. Once the process runs, the code becomes unreadable by human eyes, hence the name Uglify. Uglify has Gulp bindings you can install with

```
$ npm install --save-dev gulp-uglify
```

This installs the Gulp plugin for Uglify to your Node modules and saves an entry in the package.json file. You're now able to use this plugin within your Gulpfile, as shown in the following listing.

Listing 2.3 Uglifying JavaScript—Gulpfile.js

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});
```

Next to Gulp, you require the previously installed gulp-uglify module. This allows you to transform contents from Gulp's readable streams with the Uglify process.

You use this plugin directly after creating the readable stream and before saving it with a writeable stream. It's as easy as calling a function.

Each file gets piped through the Uglify process, transforming its contents accordingly. If you run the task with `gulp scripts` and take a good look into the dest directory, you'll see that all the perfectly clear JavaScript from earlier is now an unreadable mess. Mission accomplished!

2.3.2 Changing the file structure

When you want to load JavaScript applications over the wire, you want to make as few requests as possible (see chapter 1 for the reasons). That's why you want to combine all JavaScript files into one file. This can be done using concatenation. Concatenation combines the contents of many files into one. This new file contains all the contents from the concatenated files and needs a new name that you can define. The same goes for Gulp and the virtual file system.

In standard streams, it's usual to see the file just as a possible input source for the real data, which has to be processed. All information on the origin, like the path or filename, is lost once the stream has opened up. But because you're not just working with the contents of one or a few files, most likely with a huge amount of files, Gulp needs this information. Think of having 20 JavaScript files and wanting to minify them. You'd have to remember each filename separately and keep track of which data belongs to which file to restore a connection once the output (the minified files of the same name) must be saved.

Luckily, Gulp takes care of that for you by creating both a new input source and a data type that can be used for your streams: virtual file objects. You use a special symbol for those objects, which is shown in figure 2.11.

Once a Gulp stream is opened, all the original, physical files are wrapped in such a virtual file object and handled in the virtual file system, or *Vinyl*, as the corresponding software is called in Gulp.

Vinyl objects, the file objects of your virtual file system, contain two types of information: the path where the file originated, which becomes the file's name, as well as a stream exposing the file's contents. Those virtual files are stored in your computer's memory, known for being the fastest way to process data. There all the modifications

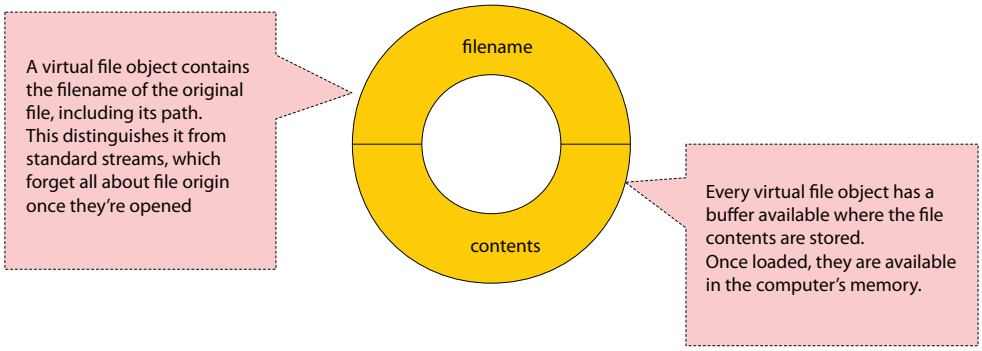


Figure 2.11 The symbol of a virtual file object. A virtual file consists mainly of two parts: data about the file's path and name, as well as a buffer containing the original contents. Both types of information are available in the computer's memory.

are done that would usually be made on your hard disk. By keeping everything in memory and not having to perform expensive read and write operations in between processes, Gulp can make changes extraordinarily fast.

You can use this virtual file system to modify the structure of your files during the Gulp task. During concatenation, you specify a new virtual file that contains all the contents from the previous stream. You just have to give it a name. See the next listing for more information.

Listing 2.4 Concatenating files—Gulpfile.js

```
var gulp = require('gulp');
var concat = require('gulp-concat');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(concat('bundle.js'))
    .pipe(gulp.dest('dist'));
});
```

❶ Require a module called gulp-concat that handles concatenation

❷ Use this module again after the readable stream is created and before the writeable stream is created

❶ This module can be installed as before with `npm install --save-dev gulp-concat`. ❷ You pipe your contents (all JavaScript files) through the `concat` process. The Concat plugin needs one parameter: the name of the new file. Internally, the Concat plugin creates a new virtual file object with the name provided by the parameter. The contents of this virtual file object are all contents from the stream files.

2.3.3 Chaining plugins

In the previous examples, you used different programs to transform the contents of a certain input set before storing the result to the hard disk. The possibilities don't end here. You can slot in any number of programs before you get to the destination, piping your data stream through multiple transformation processes.

Transformation processes are meant to do just one thing and do that one thing well. By compositing more of those processes by connecting them, you can create more advanced and sophisticated programs. To quote Doug McIlroy, who created the concept of streams back in 1964 for the Unix operating system:

> *We should have some ways of connecting programs like garden hose—screw in another segment when it becomes necessary to massage data in another way.*

Opening your data and piping it through a series of processes or subtasks is the very essence of how Gulp works. See figure 2.12, where you use both `uglify` and `concat` from the previous example in one process chain.

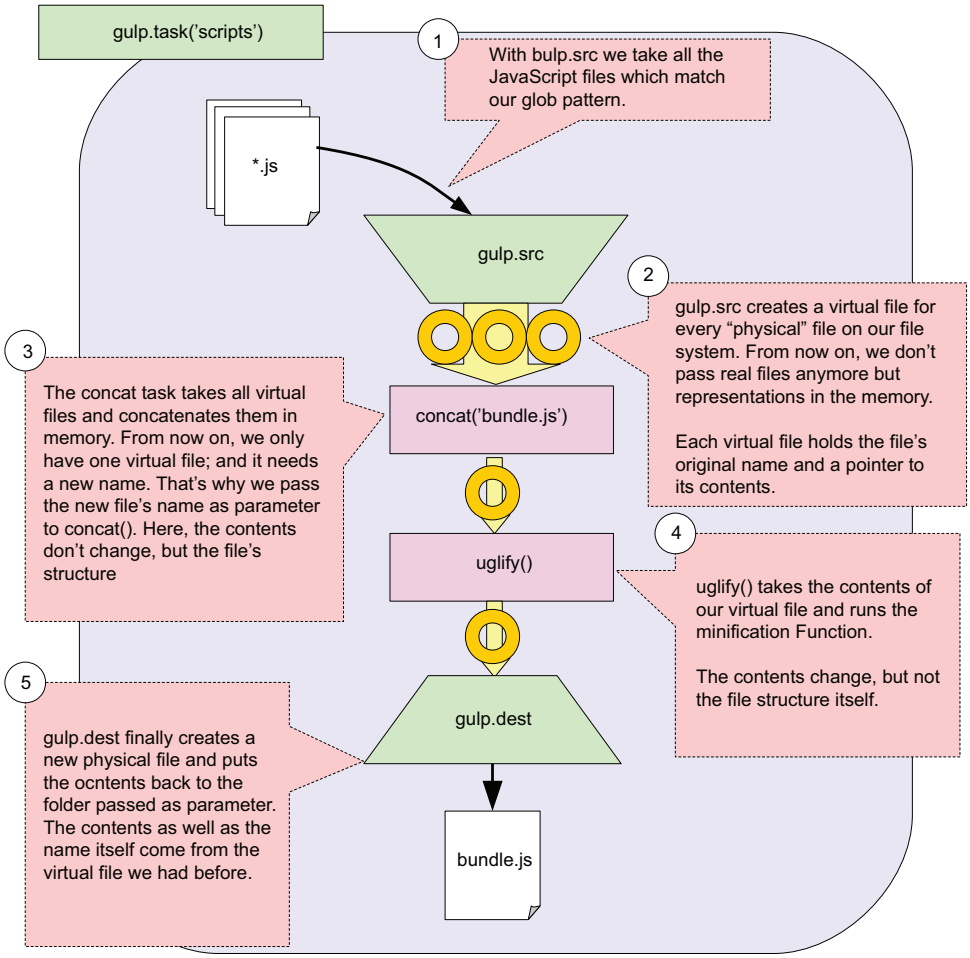


Figure 2.12 Streams applied to the virtual file system and Gulp. With `gulp.src` you select a sequence of files, promote them to the virtual file system as file objects, and access their contents for a series of transformation processes provided by the Gulp plugins.

So you're concatenating all files and then uglifying them right after. All of this is happening in the computer's memory, without costly read and write operations from the hard disk in between. This makes Gulp extremely fast and also exceptionally flexible. You can now create advanced programs that take care of multiple things, just by chaining the right plugins in order:

1. Create a `scripts` task that concatenates all files and then uglifies them.
2. Create a `styles` task that compiles the "less" files, minifies them, and then runs Autoprefixer on it to automatically add vendor prefixes.
3. Create a `test` task that does some quality checks on your JavaScript files, making sure you have a good coding style.

Vendor prefixes

Vendor prefixes are used by browser vendors to denote experimental features, where syntax or functionality is subject to change. Browser vendors use a specific abbreviation that's put before JavaScript methods and CSS properties to differentiate between other browser vendors' implementations, their own, and the final specification. For instance, `-webkit-animation` is the vendor-prefixed version of the CSS `animation` property found in browsers using the WebKit rendering engine.

The following Gulpfile takes care of your scripts and stylesheets and also does some extra testing. Install all necessary plugins as you did earlier with `npm install --save-dev <plugin-name>`.

Listing 2.5 A complete Gulpfile.js

```
var gulp        = require('gulp');
var jshint      = require('gulp-jshint');
var uglify      = require('gulp-uglify');
var concat      = require('gulp-concat');
var less        = require('gulp-less');
var minifyCSS   = require('gulp-cssnano');
var prefix      = require('gulp-autoprefixer');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(concat('main.min.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});

gulp.task('styles', function() {
  return gulp.src('app/styles/main.less')
    .pipe(less())
    .pipe(minifyCSS())
    .pipe(prefix())
    .pipe(gulp.dest('dist/styles'));
});
```

1 Require all the modules necessary for this Gulpfile

2 The script task

3 The styles task

```
gulp.task('test', function() {                          <-- 4 The test task
  return gulp.src(['app/scripts/**/*.js',
➥'!app/scripts/vendor/**/*.js'])#D
    .pipe(jshint())
    .pipe(jshint.reporter('default'))                   <-- 5 JSHint works a little differently
    .pipe(jshint.reporter('fail'));                         than the previous tasks
});
```

❶ Install each module with `npm install --save-dev <plugin-name>`. ❷ The `script` task loads all JavaScript files in the app's scripts directory and combines them into one uglified JavaScript file. ❸ The `styles` task loads one LESS main file and pipes it through three processes: LESS, CSS Minification with CSS Nano, and automatic vendor prefix inclusion with Autoprefixer. ❹ Like the `script` task, the `test` task loads all scripts, but there's a second glob excluding all the files in the vendor directory. This is because you don't want to have code style checks on third-party libraries in the vendor directory, because those files most likely have a different coding style. ❺ JSHint's output is not transformed files but a report telling you whether all style checks have been passed. You can pass this report to the reporter plugins of JSHint.

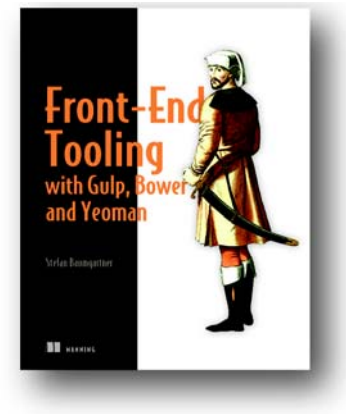
That's quite a lot that you can achieve with just a few lines of code. Running `gulp scripts`, `gulp styles`, or `gulp test` from the command line activates those specific tasks. You now have a fully functioning build file that takes care of all your assets.

2.4 Summary

In this chapter, we introduced you to our building system, Gulp:

- Gulp's runtime environment, Node.js, comes with a package manager called NPM. NPM can install Node.js modules globally to be used as a tool and locally to be used as a library. Gulp has both a command-line interface that's to be installed globally and a library that's installed locally for every project.
- Gulpfiles are build instructions written in JavaScript. They use the local Gulp installation to access an API. You can now require Node modules from within JavaScript files and use them in your code.
- Gulp's `gulp.task` API makes functions available and runnable from the command line. A call with `gulp <taskname>` executes the defined task in your file.
- `gulp.src` and `gulp.dest` create readable and writeable streams, allowing you to copy files from one place in the file system to another.
- Plugins like Concat and Uglify allow you to transform your JavaScript contents.
- Plugin chaining allows more advanced software such as a script task that does both uglification and concatenation or a styles task that takes care of running a preprocessor, using CSS minification and automatic prefixing of properties.
- Code style checks with JSHint make sure your software is well written.

With this simple Gulpfile, you can process your source files as needed.



You need a great suite of tools to minimize the time you spend on tedious non-coding tasks. Gulp is a build system that lets you run a multitude of file transforming processes with a single click. And with so many libraries working together, a dependency manager such as Bower can keep track of versions, notify you of conflicts, and in some cases even fix them. Finally, a scaffolding tool like Yeoman lets you create your applications and modules with just a single command. These tools give you an efficient workflow that speeds up the time it takes to get your applications running and deployed.

Front-End Tooling with Gulp, Bower, and Yeoman teaches you how to use and combine these popular tools to set up a customized development workflow from start to finish. Starting with the big picture of the development process, you'll see how these tools mesh together. Using patterns and examples, this in-depth book teaches you to use each tool. The second half of the book takes you deeper, showing you how to integrate and extend these tools even more in your development process. By the end of this book, you'll be skilled at using Gulp, Bower, and Yeoman and combining them to create a powerful, tailored workflow for you and your team.

What's inside

- Making advanced Gulp build files
- Creating and distributing components with Bower
- Building a distributed project generator for Yeoman
- Scaling workflows for groups

This book is suitable for front-end developers with JavaScript experience.

index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W