A JavaScript and jQuery Developer's Guide

Learning

JavaScript Design Patterns

O'REILLY®

Addy Osmani

Learning JavaScript Design Patterns

With *Learning JavaScript Design Patterns*, you'll learn how to write beautiful, structured, and maintainable JavaScript by applying classical and modern design patterns to the language. If you want to keep your code efficient, more manageable, and up-to-date with the latest best practices, this book is for you.

Explore many popular design patterns, including Modules, Observers, Facades, and Mediators. Learn how modern architectural patterns—such as MVC, MVP, and MVVM—are useful from the perspective of a modern web application developer. This book also walks you through modern module formats, how to namespace code effectively, and other essential topics.

- Learn the structure of design patterns and how they are written
- Understand different pattern categories, including creational, structural, and behavioral
- Walk through more than 20 classical and modern design patterns in JavaScript
- Use several options for writing modular code—including the Module pattern, Asyncronous Module Definition (AMD), and CommonJS
- Discover design patterns implemented in the jQuery library
- Learn popular design patterns for writing maintainable jQuery plug-ins

Addy Osmani, a Developer Programs Engineer on the Chrome team at Google, has a passion for JavaScript application architecture. He's created popular projects like TodoMVC and contributed to other open source projects such as Modernizr and jQuery. A prolific blogger (http://addyosmani.com/blog/), Addy's articles are frequently featured in *JavaScript Weekly*, *Smashing Magazine*, and many other publications.

"This book should be in every JavaScript developer's hands. It's the go-to book on JavaScript patterns that will be read and referenced many times in the future."

—Andrée Hansson
Lead Front-End Developer, presis!

"With his book, Addy Osmani makes JavaScript design patterns accessible to a larger number of developers. The sections on MV and Modern Modular patterns will help developers solidify their understanding of the techniques and libraries they are probably already using to create client-heavy web apps."*

—Eric Ferraiuolo
(@ericf), YUI, Yahoo!

US 34.99 **CAN $36.99**
ISBN: 978-1-449-33181-8
9 781449 331818 53499

Learning JavaScript Design Patterns

Addy Osmani

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Learning JavaScript Design Patterns
by Addy Osmani

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary Treseler
Production Editor: Rachel Steely
Proofreader: Linley Dolby
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Rebecca Demarest

August 2012: First Edition.

Revision History for the First Edition:

2012-08-06	First release
2012-09-14	Second release
2013-03-01	Third release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449331818* for release details.

ISBN: 978-1-449-33181-8

[LSI]

1361833802

Table of Contents

Preface

Design patterns are reusable solutions to commonly occurring problems in software design. They provide both an exciting and fascinating topic to explore in any programming language.

One reason for this is that they help us build upon the combined experience of many developers that came before us and ensure we structure our code in an optimized way, meeting the needs of problems we're attempting to solve.

Design patterns also provide us a common vocabulary to describe solutions. This can be significantly simpler than describing syntax and semantics when we're attempting to convey a way of structuring a solution in code form to others.

In this book, we will explore applying both classical and modern design patterns to the JavaScript programming language.

Target Audience

This book is targeted at professional developers wishing to improve their knowledge of design patterns and how they can be applied to the JavaScript programming language.

Some of the concepts covered (closures, prototypal inheritance) will assume a level of basic prior knowledge and understanding. If you find yourself needing to read further about these topics, a list of suggested titles is provided for convenience.

If you would like to learn how to write beautiful, structured, and organized code, I believe this is the book for you.

Credits

While many of the patterns covered in this book were implemented based on personal experience, many of them have been previously identified by the JavaScript community. This work is as such the production of the combined experience of a number of developers. Similar to Stoyan Stefanov's logical approach to preventing interruption of the narrative with credits (in *JavaScript Patterns*), I have listed credits and suggested reading for any content covered in the references section.

If any articles or links have been missed in the list of references, please accept my heartfelt apologies. If you contact me, I'll be sure to update them to include you on the list.

Reading

Although this book is targeted at both beginners and intermediate developers, a basic understanding of JavaScript fundamentals is assumed. Should you wish to learn more about the language, I am happy to recommend the following titles:

- *JavaScript: The Definitive Guide* by David Flanagan
- *Eloquent JavaScript* by Marijn Haverbeke
- *JavaScript Patterns* by Stoyan Stefanov
- *Writing Maintainable JavaScript* by Nicholas Zakas
- *JavaScript: The Good Parts* by Douglas Crockford

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning JavaScript Design Patterns* by Addy Osmani (O'Reilly). Copyright 2012 Addy Osmani, 978-1-449-33181-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/js_design_patterns*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank the passionate, talented technical reviewers who helped review and improve this book, including those from the community at large. The knowledge and enthusiasm they brought to the project was simply amazing. The official technical reviewers, tweets, and blogs are also a regular source of both ideas and inspiration and I wholeheartedly recommend checking them out.

- Nicholas Zakas (*http://nczonline.net*, @slicknet)
- Andrée Hansson (*http://andreehansson.se*, @peolanha)
- Luke Smith (*http://lucassmith.name*, @ls_n)
- Eric Ferraiuolo (*http://ericf.me/*, @ericf)
- Peter Michaux (*http://michaux.ca*, @petermichaux)
- Alex Sexton (*http://alexsexton.com*, @slexaxton)

I would also like to give a shout-out to Rebecca Murphey (*http://rebeccamurphey.com*, @rmurphey) for providing the inspiration to write this book and more importantly, continue to make it both available on GitHub and via O'Reilly.

Finally, I would like to thank my wonderful wife, Ellie, for all of her support while I was putting together this publication.

CHAPTER 1

Introduction

One of the most important aspects of writing maintainable code is being able to notice the recurring themes in that code and optimize them. This is an area where knowledge of design patterns can prove invaluable.

In the first part of this book, we will explore the history and importance of design patterns, which can really be applied to any programming language. If you're already sold on or are familiar with this history, feel free to skip to Chapter 2 to continue reading.

Design patterns can be traced back to the early work of an architect named Christopher Alexander. He would often write publications about his experience in solving design issues and how they related to buildings and towns. One day, it occurred to Alexander that when used time and time again, certain design constructs lead to a desired optimal effect.

In collaboration with Sara Ishikawa and Murray Silverstein, Alexander produced a pattern language that would help empower anyone wishing to design and build at any scale. This was published back in 1977 in a paper titled "A Pattern Language," which was later released as a complete hardcover book.

Some 30 years ago, software engineers began to incorporate the principles Alexander had written about into the first documentation about design patterns, which was to be a guide for novice developers looking to improve their coding skills. It's important to note that the concepts behind design patterns have actually been around in the programming industry since its inception, albeit in a less formalized form.

One of the first and arguably most iconic formal works published on design patterns in software engineering was a book in 1995 called *Design Patterns: Elements of Reusable Object-Oriented Software*. This was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—a group that became known as the Gang of Four (or GoF for short).

The GoF's publication is considered quite instrumental to pushing the concept of design patterns further in our field, as it describes a number of development techniques and pitfalls, as well as providing 23 core object-oriented design patterns frequently used around the world today. We will be covering these patterns in more detail in Chapter 7.

In this book, we will take a look at a number of popular JavaScript design patterns and explore why certain patterns may be more suitable for your projects than others. Remember that patterns can be applied not just to vanilla JavaScript (i.e., standard JavaScript code), but also to abstracted libraries such as jQuery and Dojo. Before we begin, let's look at the exact definition of a "pattern" in software design.

CHAPTER 2

What Is a Pattern?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design—in our case, in writing JavaScript web applications. Another way of looking at patterns is as templates for how we solve problems—ones that can be used in quite a few different situations.

So, why is it important to understand patterns and be familiar with them? Design patterns have three main benefits:

Patterns are proven solutions.
: They provide solid approaches to solving issues in software development using proven techniques that reflect the experience and insights the developers that helped define them bring to the pattern.

Patterns can be easily reused.
: A pattern usually reflects an out-of-the-box solution that can be adapted to suit our own needs. This feature makes them quite robust.

Patterns can be expressive.
: When we look at a pattern, there's generally a set structure and *vocabulary* to the solution presented that can help express rather large solutions quite elegantly.

Patterns are *not* exact solutions. It's important that we remember the role of a pattern is merely to provide us with a solution scheme. Patterns don't solve all design problems, nor do they replace good software designers, however, they *do* support them. Next, we'll take a look at some of the other advantages patterns have to offer.

- **Reusing patterns assists in preventing minor issues that can cause major problems in the application development process.** What this means is when code is built on proven patterns, we can afford to spend less time worrying about the structure of our code and more time focusing on the quality of our overall solution. This is because patterns can encourage us to code in a more structured and organized fashion, avoiding the need to refactor it for cleanliness purposes in the future.

- **Patterns can provide generalized solutions, documented in a fashion that doesn't require them to be tied to a specific problem.** This generalized approach means that, regardless of the application (and in many cases the programming language), we are working with, design patterns can be applied to improve the structure of our code.
- **Certain patterns can actually decrease the overall file-size footprint of our code by avoiding repetition.** By encouraging developers to look more closely at their solutions for areas where instant reductions in repetition can be made—e.g., reducing the number of functions performing similar processes in favor of a single generalized function—the overall size of our codebase can be decreased. This is also known as making code more *dry*.
- **Patterns add to a developers vocabulary, which makes communication faster.**
- **Patterns that are frequently used can be improved over time by harnessing the collective experiences other developers using those patterns contribute back to the design pattern community.** In some cases, this leads to the creation of entirely new design patterns, while in others it can lead to the provision of improved guidelines on how specific patterns can be best used. This can ensure that pattern-based solutions continue to become more robust than ad hoc solutions may be.

We Already Use Patterns Every Day

To understand how useful patterns can be, let's review a very simple element selection problem that the jQuery library solves for us.

Imagine that we have a script where for each DOM element found on a page with class "foo," we wish to increment a counter. What's the most efficient way to query for this collection of elements? Well, there are a few different ways this problem could be tackled:

- Select all of the elements in the page and then store references to them. Next, filter this collection and use regular expressions (or another means) to store only those with the class "foo."
- Use a modern native browser feature such as `querySelectorAll()` to select all of the elements with the class "foo."
- Use a native feature such as `getElementsByClassName()` to similarly get back the desired collection.

So, which of these options is the fastest? It's actually the third option by a factor of 8 to 10 times the alternatives. In a real-world application, however, the third option will not work in versions of Internet Explorer below 9; thus, it's necessary to use the first option when neither of the others is supported.

Developers using jQuery don't have to worry about this problem, however, as it's luckily abstracted away for us using the *Facade* pattern. As we'll review in more detail later, this pattern provides a simple set of abstracted interfaces (e.g., `$el.css()`, `$el.animate()`) to several more complex underlying bodies of code. As we've seen, this means less time having to be concerned about implementation-level details.

Behind the scenes, the library simply opts for the most optimal approach to selecting elements depending on what our current browser supports, and we just consume the abstraction layer.

We're probably all also familiar with jQuery's `$("selector")`. This is significantly easier to use for selecting HTML elements on a page versus having to manually handle opt for `getElementById()`, `getElementsByClassName()`, `getElementByTagName`, and so on.

Although we know that `querySelectorAll()` attempts to solve this problem, compare the effort involved in using jQuery's Facade interfaces versus choosing the most optimal selection paths ourselves. There's no contest! Abstractions using patterns offer real-world value.

We'll be looking at this and more design patterns later in the book.

CHAPTER 3

"Pattern"-ity Testing, Proto-Patterns, and the Rule of Three

Remember that not every algorithm, best practice, or solution represents what might be considered a complete pattern. There may be a few key ingredients here that are missing, and the pattern community is generally wary of something claiming to be one unless it has been heavily vetted. Even if something is presented to us that *appears* to meet the criteria for a pattern, it should not be considered one until it has undergone suitable periods of scrutiny and testing by others.

Looking back upon the work by Alexander once more, he claims that a pattern should be both a process and a "thing." This definition is obtuse on purpose, as he follows by saying that the process should create the "thing." This is a reason why patterns generally focus on addressing a visually identifiable structure—i.e., we should be able to visually depict the structure that results from the pattern in practice.

In studying design patterns, it's not irregular to come across the term "proto-pattern": a pattern that has not yet been known to pass the "pattern"-ity tests. Proto-patterns may result from the work of someone who has established a particular solution that is worthy of sharing with the community, but has not yet been vetted heavily due to its very young age.

Alternatively, the individual(s) sharing the pattern may not have the time or interest of going through the "pattern"-ity process and might release a short description of their proto-pattern instead. Brief descriptions or snippets of this type of pattern are known as *patlets*.

The work involved in fully documenting a qualified pattern can be quite daunting. Looking back at some of the earliest work in the field of design patterns, a pattern may be considered "good" if it does the following:

- **Solves a particular problem.** Patterns are not supposed to just capture principles or strategies. They need to capture solutions. This is one of the most essential ingredients for a good pattern.
- **Does not have an obvious solution.** We can find that problem-solving techniques often attempt to derive from well-known first principles. The best design patterns usually provide solutions to problems indirectly; this is considered a necessary approach for the most challenging problems related to design.
- **Describes a proven concept.** Design patterns require proof that they function as described, and without this proof, the design cannot be seriously considered. If a pattern is highly speculative in nature, only the brave may attempt to use it.
- **Describes a relationship.** In some cases, it may appear that a pattern describes a type of module. Although an implementation may appear this way, the official description of the pattern must describe much deeper system structures and mechanisms that explain its relationship to code.

We would be forgiven for thinking that a proto-pattern that fails to meet guidelines isn't worth learning from; however, this is far from the truth. Many proto-patterns are actually quite good. I'm not saying that all proto-patterns are worth looking at, but there are quite a few useful ones in the wild that could assist us with future projects. Use your best judgment with the above list in mind, and you'll be fine in your selection process.

One of additional requirement for a pattern to be valid is that it display some recurring phenomenon. This is often something that can be qualified in at least three key areas, referred to as the *rule of three*. To show recurrence using this rule, one must demonstrate:

Fitness of purpose
: How is the pattern considered successful?

Usefulness
: Why is the pattern considered successful?

Applicability
: Is the design worthy of being a pattern because it has wider applicability? If so, this needs to be explained.

CHAPTER 4

The Structure of a Design Pattern

You may be curious about how a pattern author might approach outlining a structure, implementation, and purpose of a new pattern. A pattern is initially presented in the form of a *rule* that establishes a relationship between:

- A context
- A system of forces that arises in that context
- A configuration that allows these forces to resolve themselves in context

With this in mind, lets now take a look at a summary of the component elements for a design pattern. A design pattern should have the following:

Pattern name

Description

Context outline
  The contexts in which the pattern is effective in responding to users' needs.

Problem statement
  A statement of the problem being addressed so we can understand the intent of the pattern.

Solution
  A description of how the user's problem is being solved in an understandable list of steps and perceptions.

Design
  A description of the pattern's design and, in particular, the user's behavior in interacting with it.

Implementation
  A guide to how the pattern would be implemented.

Illustrations
  Visual representations of classes in the pattern (e.g., a diagram).

Examples
: Implementations of the pattern in a minimal form.

Corequisites
: What other patterns may be needed to support use of the pattern being described?

Relations
: What patterns does this pattern resemble? Does it closely mimic any others?

Known usage
: Is the pattern being used in the wild? If so, where and how?

Discussions
: The team or author's thoughts on the exciting benefits of the pattern.

Design patterns are quite a powerful approach to getting all of the developers in an organization or team on the same page when creating or maintaining solutions. If you are considering working on a pattern of your own, remember that although patterns may have a heavy initial cost in the planning and write-up phases, the value returned from that investment can be quite worth it. Always research thoroughly before working on new patterns however, as you may find it more beneficial to use or build on top of existing proven patterns, rather than starting afresh.

CHAPTER 5

Writing Design Patterns

Although this book is aimed at those new to design patterns, a fundamental understanding of how a design pattern is written can offer a number of useful benefits. For starters, we can gain a deeper appreciation for the reasoning behind why a pattern is needed. We can also learn how to tell if a pattern (or proto-pattern) is up to scratch when reviewing it for our own needs.

Writing good patterns is a challenging task. Patterns not only need to (ideally) provide a substantial quantity of reference material for end users, but they also need to be able to defend why they are necessary.

Having read the previous section on *what* a pattern is, we may think that this in itself is enough to help us identify patterns we see in the wild. This is actually not completely true. It's not always clear if a piece of code we're looking at is following a set pattern or just accidentally happens to appear like it does.

When we're looking at a body of code we think may be using a pattern, we should consider writing down some of the aspects of the code that we believe fall under a particular existing pattern or set of patterns.

In many cases of pattern analysis, we can find that we're just looking at code that follows good principles and design practices that could happen to overlap with the rules for a pattern by accident. Remember: solutions in which neither interactions nor defined rules appear are *not* patterns.

If you are interested in venturing down the path of writing your own design patterns, I recommend learning from others who have already been through the process and done it well. Spend time absorbing the information from a number of different design pattern descriptions and take in what's meaningful to you.

Explore structure and semantics—this can be done by examining the interactions and context of the patterns you are interested in so you can identify the principles that assist in organizing those patterns together in useful configurations.

Once we've exposed ourselves to a wealth of information on pattern literature, we may wish to begin writing our pattern using an *existing* format and see if we can brainstorm new ideas for improving it or integrating our ideas in there.

An example of a developer that did this in recent years is Christian Heilmann, who took the existing *Module* pattern and made some fundamentally useful changes to it to create the *Revealing Module* pattern (this is one of the patterns covered later in this book).

The following are tips I would suggest if you are interested in creating a new design pattern:

How practical is the pattern?
: Ensure that the pattern describes proven solutions to recurring problems rather than just speculative solutions that haven't been qualified.

Keep best practices in mind.
: The design decisions we make should be based on principles we derive from an understanding of best practices.

Our design patterns should be transparent to the user.
: Design patterns should be entirely transparent to any type of user experience. They are primarily there to serve the developers using them and should not force changes to behavior in the user experience that would not be incurred without the use of a pattern.

Remember that originality is not key in pattern design.
: When writing a pattern, you do not need to be the original discoverer of the solutions being documented nor do you have to worry about your design overlapping with minor pieces of other patterns. If the approach is strong enough to have broad useful applicability, it has a chance of being recognized as a valid pattern.

Patterns need a strong set of examples
: A good pattern description needs to be followed by an equally strong set of examples demonstrating the successful application of the pattern. To show broad usage, examples that exhibit good design principles are ideal.

Pattern writing is a careful balance between creating a design that is general, specific, and above all, useful. Try to ensure that when writing a pattern you cover the widest possible areas of application. I hope that this brief introduction to writing patterns has given you some insights that will assist your learning process for the next sections of this book.

CHAPTER 6

Anti-Patterns

If we consider that a pattern represents a best practice, an anti-pattern represents a lesson that has been learned. The term anti-patterns was coined in 1995 by Andrew Koenig in the November C++ Report that year, inspired by the GoF's book *Design Patterns*. In Koenig's report, there are two notions of anti-patterns that are presented. Anti-patterns:

- Describe a *bad* solution to a particular problem that resulted in a bad situation occurring
- Describe *how* to get out of said situation and how to go from there to a good solution

On this topic, Alexander writes about the difficulties in achieving a good balance between good design structure and good context:

> These notes are about the process of design; the process of inventing physical things which display a new physical order, organization, form, in response to function....every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem.

While it's quite important to be aware of design patterns, it can be equally important to understand anti-patterns. Let us qualify the reason behind this. When creating an application, a project's lifecycle begins with construction; however, once you have the initial release, it needs to be maintained. The quality of a final solution will either be good or bad, depending on the level of skill and time the team has invested in it. Here, *good* and *bad* are considered in context—a "perfect" design may qualify as an anti-pattern if applied in the wrong context.

The bigger challenges happen after an application has hit production and is ready to go into maintenance mode. A developer working on such a system who hasn't worked on the application before may introduce a *bad* design into the project by accident. If said *bad* practices are created as anti-patterns, they allow developers a means to recognize these in advance so that they can avoid common mistakes that can occur; this is parallel to how design patterns provide a way for us to recognize common techniques that are *useful*.

To summarize, an anti-pattern is a bad design that is worthy of documenting. Examples of anti-patterns in JavaScript are the following:

- Polluting the global namespace by defining a large number of variables in the global context.
- Passing strings rather than functions to either `setTimeout` or `setInterval`, as this triggers the use of `eval()` internally.
- Modifying the `Object` class prototype (this is a particularly bad anti-pattern).
- Using JavaScript in an inline form as this is inflexible.
- The use of `document.write` where native DOM alternatives such as `document.createElement` are more appropriate. `document.write` has been grossly misused over the years and has quite a few disadvantages, including that if it's executed after the page has been loaded, it can actually overwrite the page we're on, while `document.createElement` does not. Visit this site for a live example of this in action. It also doesn't work with XHTML, which is another reason opting for more DOM-friendly methods such as `document.createElement` is favorable.

Knowledge of anti-patterns is critical for success. Once we are able to recognize such anti-patterns, we're able to refactor our code to negate them so that the overall quality of our solutions improves instantly.

CHAPTER 7

Categories of Design Patterns

A glossary from the well-known design book, *Domain-Driven Terms,* rightly states that:

> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities.
>
> Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation.
>
> Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ....

Design patterns can be broken down into a number of different categories. In this section, we'll review three of these categories and briefly mention a few examples of the patterns that fall into these categories before exploring specific ones in more detail.

Creational Design Patterns

Creational design patterns focus on handling object-creation mechanisms where objects are created in a manner suitable for a given situation. The basic approach to object creation might otherwise lead to added complexity in a project, while these patterns aim to solve this problem by *controlling* the creation process.

Some of the patterns that fall under this category are: Constructor, Factory, Abstract, Prototype, Singleton, and Builder.

Structural Design Patterns

Structural patterns are concerned with object composition and typically identify simple ways to realize relationships between different objects. They help ensure that when one part of a system changes, the entire structure of the system doesn't need to do the same. They also assist in recasting parts of the system that don't fit a particular purpose into those that do.

Patterns that fall under this category include: Decorator, Facade, Flyweight, Adapter, and Proxy.

Behavioral Design Patterns

Behavioral patterns focus on improving or streamlining the communication between disparate objects in a system.

Some behavioral patterns include: Iterator, Mediator, Observer, and Visitor.

CHAPTER 8
Design Pattern Categorization

In my early experiences of learning about design patterns, I personally found Table 8-1 a very useful reminder of what a number of patterns has to offer. It covers the 23 design patterns mentioned by the GoF. The original table was summarized by Elyse Nielsen back in 2004, and I've modified it where necessary to suit our discussion in this section of the book.

I recommend using this table as reference, but do remember that there are a number of additional patterns that are not mentioned here but will be discussed later in the book.

A Brief Note on Classes

Keep in mind that there will be patterns in this table that reference the concept of "classes." JavaScript is a classless language; however, classes can be simulated using functions.

The most common approach to achieving this is by defining a JavaScript function in which you then create an object using the `new` keyword. Use `this` to define new properties and methods for the object as follows:

```
// A Car "class"
function Car( model ) {

  this.model = model;
  this.color = "silver";
  this.year  = "2012";

  this.getInfo = function () {
    return this.model + " " + this.year;
  };

}
```

We can then instantiate the object using the `car` constructor we defined above like this:

```
var myCar = new Car("ford");

myCar.year = "2010";

console.log( myCar.getInfo() );
```

For more ways to define classes using JavaScript, see Stoyan Stefanov's useful post on them.

Let us now proceed to review the table.

Table 8-1.

Creational	Based on the concept of creating an object
Class	
Factory method	Makes an instance of several derived classes based on interfaced data or events
Object	
Abstract factory	Creates an instance of several families of classes without detailing concrete classes
Builder	Separates object construction from its representation; always creates the same type of object
Prototype	A fully initialized instance used for copying or cloning
Singleton	A class with only a single instance with global access points

Structural	Based on the idea of building blocks of objects
Class	
Adapter	Matches interfaces of different classes so classes can work together despite incompatible interfaces
Object	
Adapter	Matches interfaces of different classes so classes can work together despite incompatible interfaces
Bridge	Separates an object's interface from its implementation so the two can vary independently
Composite	A structure of simple and composite objects that makes the total object more than just the sum of its parts
Decorator	Dynamically adds alternate processing to objects
Facade	A single class that hides the complexity of an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing of information that is contained elsewhere
Proxy	A placeholder object representing the true object

Behavioral	Based on the way objects play and work together
Class	
Interpreter	A way to include language elements in an application to match the grammar of the intended language
Template method	Creates the shell of an algorithm in a method, then defers the exact steps to a subclass
Object	
Chain of responsibility	A way of passing a request between a chain of objects to find the object that can handle the request
Command	A way to separate the execution of a command from its invoker
Iterator	Sequentially accesses the elements of a collection without knowing the inner workings of the collection
Mediator	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other
Memento	Captures an object's internal state to be able to restore it later
Observer	A way of notifying change to a number of classes to ensure consistency between the classes
State	Alters an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class separating the selection from the implementation
Visitor	Adds a new operation to a class without changing the class

CHAPTER 9

JavaScript Design Patterns

In this chapter, we will explore JavaScript implementations of a number of both classic and modern design patterns.

Developers commonly wonder whether there is an *ideal* pattern or set of patterns they should be using in their workflow. There isn't a true single answer to this question; each script and web application we work on is likely to have its own individual needs, and we need to think about where we feel a pattern can offer real value to an implementation.

For example, some projects may benefit from the decoupling benefits offered by the Observer pattern (which reduces how dependent parts of an application are on one another), while others may simply be too small for decoupling to be a concern at all.

That said, once we have a firm grasp of design patterns and the specific problems they are best suited to, it becomes much easier to integrate them into our application architectures.

The patterns we will be exploring in this section include:

- "The Constructor Pattern" on page 22
- "The Module Pattern" on page 26
- "The Revealing Module Pattern" on page 37
- "The Singleton Pattern" on page 39
- "The Observer Pattern" on page 43
- "The Mediator Pattern" on page 60
- "The Prototype Pattern" on page 70
- "The Command Pattern" on page 73
- "The Facade Pattern" on page 75
- "The Factory Pattern" on page 78
- "The Mixin Pattern" on page 82

- "The Decorator Pattern" on page 87
- "Flyweight" on page 97

The Constructor Pattern

In classical object-oriented programming languages, a constructor is a special method used to initialize a newly created object once memory has been allocated for it. In JavaScript, as almost everything is an object, we're most often interested in *object* constructors.

Object constructors are used to create specific types of objects—both preparing the object for use and accepting arguments a constructor can use to set the values of member properties and methods when the object is first created (Figure 9-1).

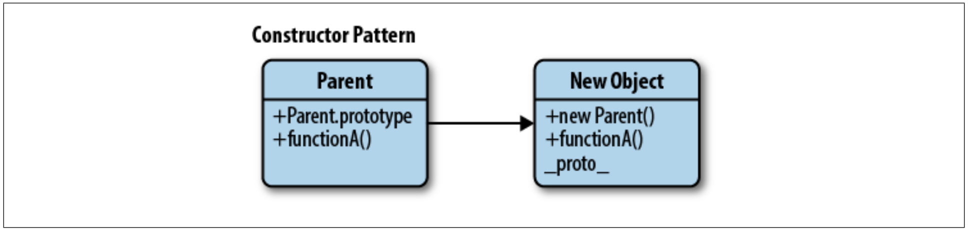


Figure 9-1. Constructor pattern

Object Creation

The two common ways to create new objects in JavaScript are as follows:

```
// Each of the following options will create a new empty object:

var newObject = {};

// or which is a shorthand for the object constructor
var newObject = new Object();
```

Where the `Object` constructor creates an object wrapper for a specific value, or where no value is passed, it will create an empty object and return it.

There are then four ways in which keys and values can be assigned to an object:

```
// ECMAScript 3 compatible approaches

// 1. Dot syntax

// Set properties
newObject.someKey = "Hello World";

// Get properties
var key = newObject.someKey;
```

```
// 2. Square bracket syntax

// Set properties
newObject["someKey"] = "Hello World";

// Get properties
var key = newObject["someKey"];



// ECMAScript 5 only compatible approaches
// For more information see: http://kangax.github.com/es5-compat-table/

// 3. Object.defineProperty

// Set properties
Object.defineProperty( newObject, "someKey", {
    value: "for more control of the property's behavior",
    writable: true,
    enumerable: true,
    configurable: true
});

// If the above feels a little difficult to read, a short-hand could
// be written as follows:

var defineProp = function ( obj, key, value ){
  config.value = value;
  Object.defineProperty( obj, key, config );
};

// To use, we then create a new empty "person" object
var person = Object.create( null );

// Populate the object with properties
defineProp( person, "car",  "Delorean" );
defineProp( person, "dateOfBirth", "1981" );
defineProp( person, "hasBeard", false );


// 4. Object.defineProperties

// Set properties
Object.defineProperties( newObject, {

  "someKey": {
    value: "Hello World",
    writable: true
  },

  "anotherKey": {
    value: "Foo bar",
    writable: false
  }

});

// Getting properties for 3. and 4. can be done using any of the
// options in 1. and 2.
```

As we will see a little later in the book, these methods can even be used for inheritance, as follows:

```
// Usage:

// Create a race car driver that inherits from the person object
var driver = Object.create( person );

// Set some properties for the driver
defineProp(driver, "topSpeed", "100mph");

// Get an inherited property (1981)
console.log( driver.dateOfBirth );

// Get the property we set (100mph)
console.log( driver.topSpeed );
```

Basic Constructors

As we saw earlier, JavaScript doesn't support the concept of classes, but it does support special constructor functions that work with objects. By simply prefixing a call to a constructor function with the keyword `new`, we can tell JavaScript we would like the function to behave like a constructor and instantiate a new object with the members defined by that function.

Inside a constructor, the keyword `this` references the new object that's being created. Revisiting object creation, a basic constructor may look as follows:

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

  this.toString = function () {
    return this.model + " has done " + this.miles + " miles";
  };
}

// Usage:

// We can create new instances of the car
var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );
```

```
// and then open our browser console to view the
// output of the toString() method being called on
// these objects
console.log( civic.toString() );
console.log( mondeo.toString() );
```

The above is a simple version of the constructor pattern, but it does suffer from some problems. One is that it makes inheritance difficult and the other is that functions such as `toString()` are redefined for each of new object created using the `Car` constructor. This isn't optimal, as the function should ideally be shared between all instances of the `Car` type.

Thankfully, as there are a number of both ES3- and ES5-compatible alternatives to constructing objects, it's trivial to work around this limitation.

Constructors with Prototypes

Functions in JavaScript have a property called a *prototype*. When we call a JavaScript constructor to create an object, all the properties of the constructor's prototype are then made available to the new object. In this fashion, multiple `Car` objects can be created that access the same prototype. We can thus extend the original example as follows:

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

}


// Note here that we are using Object.prototype.newMethod rather than
// Object.prototype so as to avoid redefining the prototype object
Car.prototype.toString = function () {
  return this.model + " has done " + this.miles + " miles";
};

// Usage:

var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

console.log( civic.toString() );
console.log( mondeo.toString() );
```

A single instance of `toString()` will now be shared between all `Car` objects.

The Module Pattern

Modules are an integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized.

In JavaScript, there are several options for implementing modules. These include:

- Object literal notation
- The Module pattern
- AMD modules
- CommonJS modules
- ECMAScript Harmony modules

We will be exploring the latter three of these options later on in the book in Chapter 11.

The Module pattern is based in part on object literals, so it makes sense to refresh our knowledge of them first.

Object Literals

In object literal notation, an object is described as a set of comma-separated name/value pairs enclosed in curly braces ({}). Names inside the object may be either strings or identifiers that are followed by a colon. There should be no comma used after the final name/value pair in the object, as this may result in errors.

```
var myObjectLiteral = {

    variableKey: variableValue,

    functionKey: function () {
      // ...
    };
};
```

Object literals don't require instantiation using the `new` operator, but shouldn't be used at the start of a statement, as the opening { may be interpreted as the beginning of a block. Outside of an object, new members may be added to the object literal using assignment as follows: `myModule.property = "someValue";`

Below, we can see a more complete example of a module defined using object literal notation:

```
var myModule = {

  myProperty: "someValue",

  // object literals can contain properties and methods.
  // e.g we can define a further object for module configuration:
  myConfig: {
    useCaching: true,
    language: "en"
  },

  // a very basic method
  myMethod: function () {
    console.log( "Where in the world is Paul Irish today?" );
  },

  // output a value based on the current configuration
  myMethod2: function () {
    console.log( "Caching is:" + ( this.myConfig.useCaching ) ? "enabled" : "disabled" );
  },

  // override the current configuration
  myMethod3: function( newConfig ) {

    if ( typeof newConfig === "object" ) {
      this.myConfig = newConfig;
      console.log( this.myConfig.language );
    }
  }
};

// Outputs: Where in the world is Paul Irish today?
myModule.myMethod();

// Outputs: enabled
myModule.myMethod2();

// Outputs: fr
myModule.myMethod3({
  language: "fr",
  useCaching: false
});
```

Using object literals can assist in encapsulating and organizing your code. Rebecca Murphey has previously written about this topic in depth, should you wish to read into object literals further.

That said, if we're opting for this technique, we may be equally as interested in the Module pattern. It still uses object literals but only as the return value from a scoping function.

The Module Pattern

The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.

In JavaScript, the Module pattern is used to further *emulate* the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope. What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page (Figure 9-2).

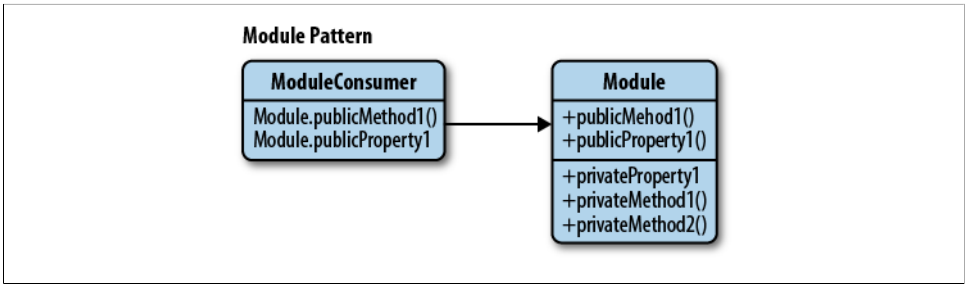


Figure 9-2. Module pattern

Privacy

The Module pattern encapsulates "privacy" state and organization using closures. It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface. With this pattern, only a public API is returned, keeping everything else within the closure private.

This gives us a clean solution for shielding logic doing the heavy lifting while only exposing an interface we would like other parts of our application to use. The pattern is quite similar to an immediately-invoked functional expression[1], except that an object is returned rather than a function.

It should be noted that there isn't really an explicitly true sense of "privacy" inside JavaScript because unlike some traditional languages, it doesn't have access modifiers. Variables can't technically be declared as being public nor private, and so we use function scope to simulate this concept. Within the Module pattern, variables or methods declared are only available inside the module itself, thanks to closure. Variables or methods defined within the returning object, however, are available to everyone.

History

From a historical perspective, the Module pattern was originally developed by a number of people, including Richard Cornford in 2003. It was later popularized by Douglas Crockford in his lectures. In addition, if you've ever played with Yahoo's YUI library, some of its features may appear quite familiar, because the Module pattern was a strong influence for YUI when creating their components.

1. IIFE. See "Namespacing Patterns" on page 202 for more on this.

Examples

Let's begin looking at an implementation of the Module pattern by creating a module that is self-contained.

```
var testModule = (function () {

  var counter = 0;

  return {

    incrementCounter: function () {
      return ++counter;
    },

    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };

})();

// Usage:

// Increment our counter
testModule.incrementCounter();

// Check the counter value and reset
// Outputs: 1
testModule.resetCounter();
```

Here, other parts of the code are unable to directly read the value of our `incrementCounter()` or `resetCounter()`. The `counter` variable is actually fully shielded from our global scope so it acts just like a private variable would—its existence is limited to within the module's closure so that the only code able to access its scope, are our two functions. Our methods are effectively namespaced so in the test section of our code, we need to prefix any calls with the name of the module (e.g., `testModule`).

When working with the Module pattern, we may find it useful to define a simple template for getting started with it. Here's one that covers namespacing, public, and private variables:

```
var myNamespace = (function () {


  // A private counter variable
  var myPrivateVar = 0;

  // A private function which logs any arguments
  var myPrivateMethod = function( foo ) {
      console.log( foo );
  };
```

```
    return {

      // A public variable
      myPublicVar: "foo",

      // A public function utilizing privates
      myPublicFunction: function( bar ) {

        // Increment our private counter
        myPrivateVar++;

        // Call our private method using bar
        myPrivateMethod( bar );

      }
    };

  })();
```

Looking at another example, we can see a shopping basket implemented using this pattern. The module itself is completely self-contained in a global variable called `basketModule`. The `basket` array in the module is kept private, so other parts of our application are unable to directly read it. It only exists with the module's closure, so the only methods able to access it are those with access to its scope (i.e., `addItem()`, `getItem()`, etc.).

```
  var basketModule = (function () {

    // privates

    var basket = [];

    function doSomethingPrivate() {
      //...
    }

    function doSomethingElsePrivate() {
      //...
    }

    // Return an object exposed to the public
    return {

      // Add items to our basket
      addItem: function( values ) {
        basket.push(values);
      },

      // Get the count of items in the basket
      getItemCount: function () {
        return basket.length;
      },

      // Public alias to a  private function
      doSomething: doSomethingPrivate,
      // Get the total value of items in the basket
      getTotal: function () {

        var itemCount = this.getItemCount(),
            total = 0;

        while (itemCount--) {
          total += basket[itemCount].price;
        }

        return total;
      }
    };
  }());
```

Inside the module, you may have noticed that we return an `object`. This gets automatically assigned to `basketModule` so that we can interact with it as follows:

```
// basketModule returns an object with a public API we can use

basketModule.addItem({
  item: "bread",
  price: 0.5
});

basketModule.addItem({
  item: "butter",
  price: 0.3
});

// Outputs: 2
console.log( basketModule.getItemCount() );

// Outputs: 0.8
console.log( basketModule.getTotal() );

// However, the following will not work:

// Outputs: undefined
// This is because the basket itself is not exposed as a part of our
// the public API
console.log( basketModule.basket );

// This also won't work as it only exists within the scope of our
// basketModule closure, but not the returned public object
console.log( basket );
```

The methods above are effectively namespaced inside `basketModule`.

Notice how the scoping function in the above basket module is wrapped around all of our functions, which we then call and immediately store the return value of. This has a number of advantages including:

- The freedom to have private functions that can only be consumed by our module. As they aren't exposed to the rest of the page (only our exported API is), they're considered truly private.
- Given that functions are declared normally and are named, it can be easier to show call stacks in a debugger when we're attempting to discover what function(s) threw exceptions.
- As T.J. Crowder has pointed out in the past, it also enables us to return different functions depending on the environment. In the past, I've seen developers use this to perform UA testing to provide a codepath in their module specific to IE, but we can easily opt for feature detection these days to achieve a similar goal.

Module Pattern Variations

Import mixins

This variation of the pattern demonstrates how globals (e.g., `jQuery`, `Underscore`) can be passed in as arguments to our module's anonymous function. This effectively allows us to *import* them and locally alias them as we wish.

```
// Global module
var myModule = (function ( jQ, _ ) {

    function privateMethod1(){
        jQ(".container").html("test");
    }

    function privateMethod2(){
      console.log( _.min([10, 5, 100, 2, 1000]) );
    }

    return{
        publicMethod: function(){
            privateMethod1();
        }
    };

// Pull in jQuery and Underscore
}( jQuery, _ ));

myModule.publicMethod();
```

Exports

This next variation allows us to declare globals without consuming them and could similarly support the concept of global imports seen in the last example.

```
// Global module
var myModule = (function () {

    // Module object
```

```
  var module = {},
    privateVariable = "Hello World";

  function privateMethod() {
    // ...
  }

  module.publicProperty = "Foobar";
  module.publicMethod = function () {
    console.log( privateVariable );
  };

  return module;

}());
```

Toolkit and framework-specific module pattern implementations

Dojo. Dojo provides a convenience method for working with objects called `dojo.setObject()`. This takes as its first argument a dot-separated string such as `myObj.parent.child`, which refers to a property called `child` within an object `parent` defined inside `myObj`. Using `setObject()` allows us to set the value of children, creating any of the intermediate objects in the rest of the path passed if they don't already exist.

For example, if we wanted to declare `basket.core` as an object of the `store` namespace, this could be achieved as follows using the traditional way:

```
var store = window.store || {};

if ( !store["basket"] ) {
  store.basket = {};
}

if ( !store.basket["core"] ) {
  store.basket.core = {};
}

store.basket.core = {
  // ...rest of our logic
};
```

Or as follows using Dojo 1.7 (AMD-compatible version) and above:

```
require(["dojo/_base/customStore"], function( store ){

  // using dojo.setObject()
  store.setObject( "basket.core", (function() {

      var basket = [];

      function privateMethod() {
          console.log(basket);
      }
```

```
            return {
                publicMethod: function(){
                        privateMethod();
                }
            };

        }()));

    });
```

For more information on `dojo.setObject()`, see the official documentation.

ExtJS. For those using Sencha's ExtJS, you're in for some luck, as the official documentation incorporates examples that demonstrate how to correctly use the Module pattern with the framework.

Here we can see an example of how to define a namespace that can then be populated with a module containing both a private and public API. With the exception of some semantic differences, it's quite close to how the Module pattern is implemented in vanilla JavaScript:

```
    // create namespace
    Ext.namespace("myNameSpace");

    // create application
    myNameSpace.app = function () {

      // do NOT access DOM from here; elements don't exist yet
      // private variables

      var btn1,
          privVar1 = 11;

      // private functions
      var btn1Handler = function ( button, event ) {
          console.log( "privVar1", privVar1 );
          console.log( "this.btn1Text=" + this.btn1Text );
        };

      // public space
      return {
        // public properties, e.g. strings to translate
        btn1Text: "Button 1",

        // public methods
        init: function () {

          if ( Ext.Ext2 ) {

            btn1 = new Ext.Button({
              renderTo: "btn1-ct",
              text: this.btn1Text,
              handler: btn1Handler
            });
          } else {

            btn1 = new Ext.Button( "btn1-ct", {
              text: this.btn1Text,
              handler: btn1Handler
            });

          }
        }
      };
    }();
```

YUI. Similarly, we can implement the Module pattern when building applications using YUI3. The following example is heavily based on the original YUI Module pattern implementation by Eric Miraglia, but again, isn't vastly different from the vanilla JavaScript version:

```
Y.namespace( "store.basket" ) = (function () {

    var myPrivateVar, myPrivateMethod;

    // private variables:
    myPrivateVar = "I can be accessed only within Y.store.basket.";

    // private method:
    myPrivateMethod = function () {
        Y.log( "I can be accessed only from within YAHOO.store.basket" );
    }

    return {
        myPublicProperty: "I'm a public property.",

        myPublicMethod: function () {
            Y.log( "I'm a public method." );

            // Within basket, I can access "private" vars and methods:
            Y.log( myPrivateVar );
            Y.log( myPrivateMethod() );

            // The native scope of myPublicMethod is store so we can
            // access public members using "this":
            Y.log( this.myPublicProperty );
        }
    };

})();
```

jQuery. There are a number of ways in which jQuery code unspecific to plug-ins can be wrapped inside the Module pattern. Ben Cherry previously suggested an implementation in which a function wrapper is used around module definitions in the event of there being a number of commonalities between modules.

In the following example, a `library` function is defined that declares a new library and automatically binds up the `init` function to `document.ready` when new libraries (i.e., modules) are created.

```
function library( module ) {

  $( function() {
    if ( module.init ) {
      module.init();
    }
  });

  return module;
}

var myLibrary = library(function () {

  return {
    init: function () {
      // module implementation
    }
  };
}());
```

Advantages

We've seen why the Singleton pattern can be useful, but why is the Module pattern a good choice? For starters, it's a lot cleaner for developers coming from an object-oriented background than the idea of true encapsulation, at least from a JavaScript perspective.

Second, it supports private data—so, in the Module pattern, public parts of our code are able to touch the private parts, however the outside world is unable to touch the class's private parts (thanks to David Engfer for the joke).

Disadvantages

The disadvantages of the Module pattern are that, as we access both public and private members differently, when we wish to change visibility, we actually have to make changes to each place the member was used.

We also can't access private members in methods that are added to the object at a later point. That said, in many cases, the Module pattern is still quite useful and, when used correctly, certainly has the potential to improve the structure of our application.

Other disadvantages include the inability to create automated unit tests for private members and additional complexity when bugs require hot fixes. It's simply not possible to patch privates. Instead, one must override all public methods that interact with the buggy privates. Developers can't easily extend privates either, so it's worth remembering privates are not as flexible as they may initially appear.

For further reading on the Module pattern, see Ben Cherry's excellent in-depth article.

The Revealing Module Pattern

Now that we're a little more familiar with the Module pattern, let's take a look at a slightly improved version—Christian Heilmann's Revealing Module pattern.

The Revealing Module pattern came about as Heilmann was frustrated with the fact that he had to repeat the name of the main object when he wanted to call one public method from another or access public variables. He also disliked the Module pattern's requirement of having to switch to object literal notation for the things he wished to make public.

The result of his efforts was an updated pattern in which he could simply define all functions and variables in the private scope and return an anonymous object with pointers to the private functionality he wished to reveal as public.

An example of how to use the Revealing Module pattern is as follows:

```
var myRevealingModule = function () {

        var privateVar = "Ben Cherry",
            publicVar  = "Hey there!";

        function privateFunction() {
            console.log( "Name:" + privateVar );
        }

        function publicSetName( strName ) {
            privateVar = strName;
        }

        function publicGetName() {
            privateFunction();
        }


        // Reveal public pointers to
        // private functions and properties

        return {
            setName: publicSetName,
            greeting: publicVar,
            getName: publicGetName
        };

    }();

myRevealingModule.setName( "Paul Kinlan" );
```

The pattern can also be used to reveal private functions and properties with a more specific naming scheme if you would prefer:

```
var myRevealingModule = function () {

        var privateCounter = 0;

        function privateFunction() {
            privateCounter++;
        }

        function publicFunction() {
            publicIncrement();
        }

        function publicIncrement() {
            privateFunction();
        }

        function publicGetCount(){
          return privateCounter;
        }

        // Reveal public pointers to
        // private functions and properties

       return {
            start: publicFunction,
            increment: publicIncrement,
            count: publicGetCount
        };

    }();

myRevealingModule.start();
```

Advantages

This pattern allows the syntax of our scripts to be more consistent. It also makes it easier to tell at the end of the module which of our functions and variables may be accessed publicly, which eases readability.

Disadvantages

A disadvantage of this pattern is that if a private function refers to a public function, that public function can't be overridden if a patch is necessary. This is because the private function will continue to refer to the private implementation, and the pattern doesn't apply to public members, only to functions.

Public object members that refer to private variables are also subject to the no-patch rule.

As a result of this, modules created with the Revealing Module pattern may be more fragile than those created with the original Module pattern, so care should be taken during usage.

The Singleton Pattern

The Singleton pattern is thus known because it restricts instantiation of a class to a single object. Classically, the Singleton pattern can be implemented by creating a class with a method that creates a new instance of the class if one doesn't exist. In the event of an instance already existing, it simply returns a reference to that object. Singletons differ from static *classes* (or objects) as we can delay their initialization, generally because they require some information that may not be available during initialization time. For code that is unaware of a previous reference to them, they do not provide a method for easy retrieval. This is because it is neither the object nor "class" that's returned by a Singleton; it's a structure. Think of how closured variables aren't actually closures—the function scope that provides the closure is the closure. In JavaScript, Singletons serve as a shared resource namespace which isolate implementation code from the global namespace so as to provide a single point of access for functions. We can implement a Singleton as follows:

```
var mySingleton = (function () {

  // Instance stores a reference to the Singleton
  var instance;

  function init() {

    // Singleton

    // Private methods and variables
    function privateMethod(){
        console.log( "I am private" );
    }

    var privateVariable = "Im also private";

    var privateRandomNumber = Math.random();

    return {

      // Public methods and variables
      publicMethod: function () {
        console.log( "The public can see me!" );
      },

      publicProperty: "I am also public"

      getRandomNumber: function() {
        return privateRandomNumber;
      }

    };

  };

  return {
    // Get the Singleton instance if one exists
    // or create one if it doesn't
    getInstance: function () {

      if ( !instance ) {
        instance = init();
      }

      return instance;
    }

  };

})();

var myBadSingleton = (function () {

  // Instance stores a reference to the Singleton
  var instance;

  function init() {

    // Singleton

    var privateRandomNumber = Math.random();

    return {

      getRandomNumber: function() {
        return privateRandomNumber;
      }

    };

  };

  return {

    // Always create a new Singleton instance
    getInstance: function () {

      instance = init();

      return instance;
    }

  };

})();


// Usage:

var singleA = mySingleton.getInstance();
var singleB = mySingleton.getInstance();
console.log( singleA.getRandomNumber() === singleB.getRandomNumber() ); // true

var badSingleA = myBadSingleton.getInstance();
var badSingleB = myBadSingleton.getInstance();
console.log( badSingleA.getRandomNumber() !== badSingleB.getRandomNumber() ); // true
```

What makes the Singleton is the global access to the instance (generally through `MySingleton.getInstance()`) as we don't (at least in static languages) call a new `MySingleton()` directly. This is, however, possible in JavaScript. In the GoF book, the *applicability* of the Singleton pattern is described as follows:

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- The sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

The second of these points refers to a case where we might need code, such as:

```
mySingleton.getInstance = function(){
  if ( this._instance == null ) {
    if ( isFoo() ) {
       this._instance = new FooSingleton();
    } else {
       this._instance = new BasicSingleton();
    }
  }
  return this._instance;
};
```

Here, `getInstance` becomes a little like a Factory method and we don't need to update each point in our code when accessing it. `FooSingleton` (above) would be a subclass of `BasicSingleton` and would implement the same interface.

Why is deferring execution considered important for a Singleton?:

> In C++, it serves as isolation from the unpredictability of the dynamic initialization order, returning control to the programmer.

It is important to note the difference between a static instance of a class (object) and a Singleton: while a Singleton can be implemented as a static instance, it can also be constructed lazily, without the use of resources or memory, until the static instance is needed.

If we have a static object that can be initialized directly, we need to ensure the code is always executed in the same order (e.g., in case `objCar` needs `objWheel` during its initialization), and this doesn't scale when you have a large number of source files.

Both Singletons and static objects are useful, but they shouldn't be overused in the same way that we shouldn't overuse other patterns.

In practice, the Singleton pattern is useful when exactly one object is needed to coordinate others across a system. Here, you can see the pattern being used in this context:

```
var SingletonTester = (function () {

  // options: an object containing configuration options for the singleton
  // e.g var options = { name: "test", pointX: 5};
  function Singleton( options )  {

    // set options to the options supplied
    // or an empty object if none are provided
    options = options || {};

    // set some properties for our singleton
    this.name = "SingletonTester";

    this.pointX = options.pointX || 6;

    this.pointY = options.pointY || 10;

  }

  // our instance holder
  var instance;

  // an emulation of static variables and methods
  var _static  = {

    name:  "SingletonTester",

    // Method for getting an instance. It returns
    // a singleton instance of a singleton object
    getInstance:  function( options ) {
      if( instance  ===  undefined )  {
        instance = new Singleton( options );
      }

      return  instance;

    }
  };

  return  _static;

})();

var singletonTest  =  SingletonTester.getInstance({
  pointX:  5
});

// Log the output of pointX just to verify it is correct
// Outputs: 5
console.log( singletonTest.pointX );
```

While the Singleton has valid uses, often when we find ourselves needing it in JavaScript, it's a sign that we may need to reevaluate our design.

The presence of the Singleton is often an indication that modules in a system are either tightly coupled or that logic is overly spread across multiple parts of a code base. Singletons can be more difficult to test due to issues ranging from hidden dependencies to difficulty in creating multiple instances, difficulty in stubbing dependencies, etc.

Miller Medeiros has previously recommended this excellent article for further reading on the Singleton and its various issues. Also recommended reading are the comments to this article, discussing how Singletons can increase tight coupling. I'm happy to second these recommendations, as both pieces raise many important points about this pattern.

The Observer Pattern

The Observer is a design pattern in which an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to state (Figure 9-3).

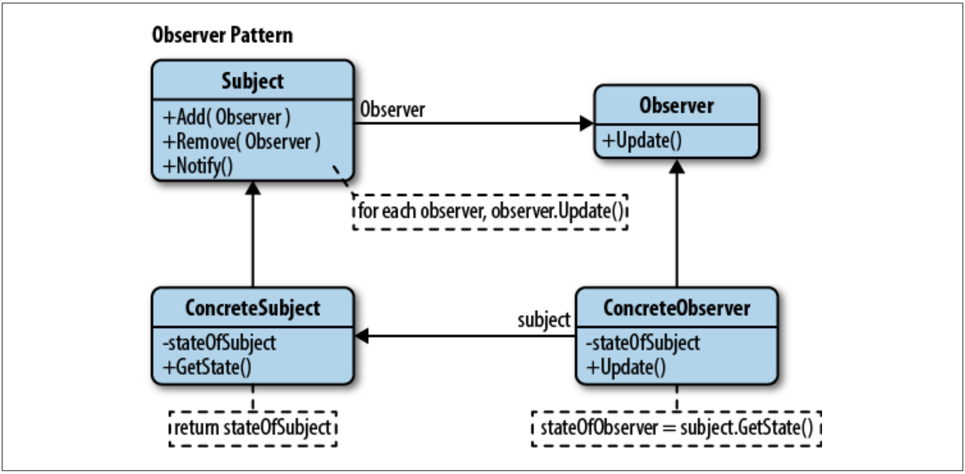


Figure 9-3. Observer pattern

When a subject needs to notify observers about something interesting happening, it broadcasts a notification to the observers (which can include specific data related to the topic of the notification).

When we no longer wish for a particular observer to be notified of changes by the subject it is registered with, the subject can remove it from the list of observers.

It's often useful to refer back to published definitions of design patterns that are language agnostic to get a broader sense of their usage and advantages over time. The definition of the Observer pattern provided in the GoF book, *Design Patterns: Elements of Reusable Object-Oriented Software*, is:

> One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves. When something changes in our subject that the observer may be interested in, a notify message is sent which calls the update method in each observer. When the observer is no longer interested in the subject's state, they can simply detach themselves.

We can now expand on what we've learned to implement the Observer pattern with the following components:

Subject
: Maintains a list of observers, facilitates adding or removing observers

Observer
: Provides an update interface for objects that need to be notified of a Subject's changes of state

`ConcreteSubject`
: Broadcasts notifications to Observers on changes of state, stores the state of `ConcreteObservers`

`ConcreteObserver`
: Stores a reference to the `ConcreteSubject`, implements an update interface for the Observer to ensure state is consistent with the Subject's

First, let's model the list of dependent Observers a subject may have:

```
function ObserverList(){
  this.observerList = [];
}

ObserverList.prototype.Add = function( obj ){
  return this.observerList.push( obj );
};

ObserverList.prototype.Empty = function(){
  this.observerList = [];
};

ObserverList.prototype.Count = function(){
  return this.observerList.length;
};


ObserverList.prototype.Get = function( index ){
  if( index > -1 && index < this.observerList.length ){
    return this.observerList[ index ];
  }
};

ObserverList.prototype.Insert = function( obj, index ){
  var pointer = -1;

  if( index === 0 ){
    this.observerList.unshift( obj );
    pointer = index;
  }else if( index === this.observerList.length ){
    this.observerList.push( obj );
    pointer = index;
  }

  return pointer;
};

ObserverList.prototype.IndexOf = function( obj, startIndex ){
  var i = startIndex, pointer = -1;

  while( i < this.observerList.length ){
    if( this.observerList[i] === obj ){
      pointer = i;
    }
  }

i++;

i.e

  while( i < this.observerList.length ){
    if( this.observerList[i] === obj ){
      pointer = i;
    }
    i++;
  }

  return pointer;
};


ObserverList.prototype.RemoveIndexAt() = function( index ){
  if( index === 0 ){
    this.observerList.shift();
  }else if( index === this.observerList.length -1 ){
    this.observerList.pop();
  }
};


// Extend an object with an extension
function extend( extension, obj ){
  for ( var key in extension ){
    obj[key] = extension[key];
  }
}
```

Next, let's model the Subject and the ability to add, remove, or notify observers on the observer list.

```
function Subject(){
  this.observers = new ObserverList();
}

Subject.prototype.AddObserver = function( observer ){
  this.observers.Add( observer );
};

Subject.prototype.RemoveObserver = function( observer ){
  this.observers.RemoveAt( this.observers.IndexOf( observer, 0 ) );
};

Subject.prototype.Notify = function( context ){
  var observerCount = this.observers.Count();
  for(var i=0; i < observerCount; i++){
    this.observers.Get(i).Update( context );
  }
};
```

We then define a skeleton for creating new Observers. The `Update` functionality here will be overwritten later with custom behavior.

```
// The Observer
function Observer(){
  this.Update = function(){
    // ...
  };
}
```

In our sample application using the above Observer components, we now define:

- A button for adding new observable checkboxes to the page
- A control checkbox, which will act as a subject, notifying other checkboxes they should be checked
- A container for the new checkboxes being added

We then define `ConcreteSubject` and `ConcreteObserver` handlers for both adding new observers to the page and implementing the updating interface. See below for inline comments on what these components do in the context of our example.

Here is the HTML code:

```
<button id="addNewObserver">Add New Observer checkbox</button>
<input id="mainCheckbox" type="checkbox"/>
<div id="observersContainer"></div>
```

Here is a sample script:

```
// References to our DOM elements

var controlCheckbox = document.getElementById( "mainCheckbox" ),
  addBtn = document.getElementById( "addNewObserver" ),
  container = document.getElementById( "observersContainer" );


// Concrete Subject

// Extend the controlling checkbox with the Subject class
extend( new Subject(), controlCheckbox );
```

```
// Clicking the checkbox will trigger notifications to its observers
controlCheckbox["onclick"] = new Function( "controlCheckbox.Notify(controlCheckbox.checked)" );


addBtn["onclick"] = AddNewObserver;

// Concrete Observer

function AddNewObserver(){

  // Create a new checkbox to be added
  var check  = document.createElement( "input" );
  check.type = "checkbox";

  // Extend the checkbox with the Observer class
  extend( new Observer(), check );

  // Override with custom update behaviour
  check.Update = function( value ){
    this.checked = value;
  };

  // Add the new observer to our list of observers
  // for our main subject
  controlCheckbox.AddObserver( check );

  // Append the item to the container
  container.appendChild( check );
}
```

In this example, we looked at how to implement and utilize the Observer pattern, covering the concepts of a Subject, Observer, `ConcreteSubject`, and `ConcreteObserver`.

Differences Between the Observer and Publish/Subscribe Pattern

While the Observer pattern is useful to be aware of, quite often in the JavaScript world, we'll find it commonly implemented using a variation known as the Publish/Subscribe pattern. While very similar, there are differences between these patterns worth noting.

The Observer pattern requires that the observer (or object) wishing to receive topic notifications must subscribe this interest to the object firing the event (the subject), as seen in Figure 9-4.

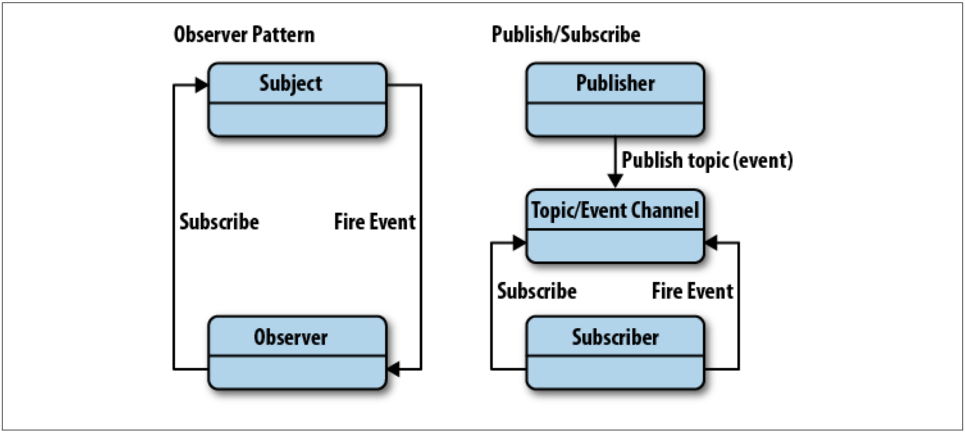


Figure 9-4. Publish/Subscribe

The Publish/Subscribe pattern however uses a topic/event channel that sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application specific events that can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

This differs from the Observer pattern as it allows any subscriber implementing an appropriate event handler to register for and receive topic notifications broadcast by the publisher.

Here is an example of how one might use the Publish/Subscribe pattern if provided with a functional implementation powering `publish()`, `subscribe()`, and `unsubscribe()` behind the scenes:

```
// A very simple new mail handler

// A count of the number of messages received
var mailCounter = 0;

// Initialize subscribers that will listen out for a topic
// with the name "inbox/newMessage".

// Render a preview of new messages
var subscriber1 = subscribe( "inbox/newMessage", function( topic, data ) {

  // Log the topic for debugging purposes
  console.log( "A new message was received: ", topic );

  // Use the data that was passed from our subject
  // to display a message preview to the user
  $( ".messageSender" ).html( data.sender );
  $( ".messagePreview" ).html( data.body );

});
```

```
// Here's another subscriber using the same data to perform
// a different task.

// Update the counter displaying the number of new
// messages received via the publisher

var subscriber2 = subscribe( "inbox/newMessage", function( topic, data ) {

  $('.newMessageCounter').html( mailCounter++ );

});

publish( "inbox/newMessage", [{
  sender:"hello@google.com",
  body: "Hey there! How are you doing today?"
}]);

// We could then at a later point unsubscribe our subscribers
// from receiving any new topic notifications as follows:
// unsubscribe( subscriber1,  );
// unsubscribe( subscriber2 );
```

The general idea here is the promotion of loose coupling. Rather than single objects calling on the methods of other objects directly, they instead subscribe to a specific task or activity of another object and are notified when it occurs.

Advantages

The Observer and Publish/Subscribe patterns encourage us to think hard about the relationships between different parts of our application. They also help us identify layers containing direct relationships that could be replaced with sets of subjects and observers. This effectively could be used to break down an application into smaller, more loosely coupled blocks to improve code management and potential for reuse.

Further motivation behind using the Observer pattern is where we need to maintain consistency between related objects without making classes tightly coupled. For example, when an object needs to be able to notify other objects without making assumptions regarding those objects.

Dynamic relationships can exist between observers and subjects when using either pattern. This provides a great deal of flexibility that may not be as easy to implement when disparate parts of our application are tightly coupled.

While it may not always be the best solution to every problem, these patterns remain one of the best tools for designing decoupled systems and should be considered an important tool in any JavaScript developer's utility belt.

Disadvantages

Consequently, some of the issues with these patterns actually stem from their main benefits. In Publish/Subscribe, by decoupling publishers from subscribers, it can sometimes become difficult to obtain guarantees that particular parts of our applications are functioning as we may expect.

For example, publishers may make an assumption that one or more subscribers are listening to them. Say that we're using such an assumption to log or output errors regarding some application process. If the subscriber performing the logging crashes (or for some reason fails to function), the publisher won't have a way of seeing this due to the decoupled nature of the system.

Another drawback of the pattern is that subscribers are quite ignorant to the existence of each other and are blind to the cost of switching publishers. Due to the dynamic relationship between subscribers and publishers, the update dependency can be difficult to track.

Publish/Subscribe Implementations

Publish/Subscribe fits in very well in JavaScript ecosystems, largely because at the core, ECMAScript implementations are event driven. This is particularly true in browser environments, as the DOM uses events as its main interaction API for scripting.

That said, neither ECMAScript nor DOM provide core objects or methods for creating custom events systems in implementation code (with the exception of perhaps the DOM3 CustomEvent, which is bound to the DOM and is thus not generically useful).

Luckily, popular JavaScript libraries such as Dojo, jQuery (custom events), and YUI already have utilities that can assist in easily implementing a Publish/Subscribe system with very little effort. Here we can see some examples of this:

```
// Publish

// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
$( el ).trigger( "/login", [{username:"test", userData:"test"}] );

// Dojo: dojo.publish("channel", [arg1, arg2, arg3] );
dojo.publish( "/login", [{username:"test", userData:"test"}] );

// YUI: el.publish("channel", [arg1, arg2, arg3]);
el.publish( "/login", {username:"test", userData:"test"} );


// Subscribe

// jQuery: $(obj).on( "channel", [data], fn );
$( el ).on( "/login", function( event ){...} );

// Dojo: dojo.subscribe( "channel", fn);
var handle = dojo.subscribe( "/login", function(data){..} );
```

```
// YUI: el.on("channel", handler);
el.on( "/login", function( data ){...} );


// Unsubscribe

// jQuery: $(obj).off( "channel" );
$( el ).off( "/login" );

// Dojo: dojo.unsubscribe( handle );
dojo.unsubscribe( handle );

// YUI: el.detach("channel");
el.detach( "/login" );
```

For those wishing to use the Publish/Subscribe pattern with vanilla JavaScript (or another library), AmplifyJS includes a clean, library-agnostic implementation that can be used with any library or toolkit. Radio.js (*http://radio.uxder.com/*), PubSubJS (*https://github.com/mroderick/PubSubJS*), or Pure JS PubSub by Peter Higgins (*https://github.com/phiggins42/bloody-jquery-plugins/blob/*) are similar alternatives worth checking out.

jQuery developers in particular have quite a few other options and can opt to use one of the many well-developed implementations ranging from Peter Higgins's jQuery plug-in to Ben Alman's (optimized) Pub/Sub jQuery gist on GitHub. Links to just a few of these can be found below.

- Ben Alman's Pub/Sub gist

  https://gist.github.com/661855 (recommended)
- Rick Waldron's jQuery-core style take on the above

  https://gist.github.com/705311
- Peter Higgins' plug-in *http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js*.
- AppendTo's Pub/Sub in AmplifyJS

  http://amplifyjs.com
- Ben Truyman's gist

  https://gist.github.com/826794

So that we are able to get an appreciation for how many of the vanilla JavaScript implementations of the Observer pattern might work, let's take a walkthrough of a minimalist version of Publish/Subscribe I released on GitHub under a project called pubsubz. This demonstrates the core concepts of subscribe, and publish, as well as the concept of unsubscribing.

I've opted to base our examples on this code as it sticks closely to both the method signatures and approach of implementation I would expect to see in a JavaScript version of the classic Observer pattern.

A Publish/Subscribe implementation

```
var pubsub = {};

(function(q) {

    var topics = {},
        subUid = -1;

    // Publish or broadcast events of interest
    // with a specific topic name and arguments
    // such as the data to pass along
    q.publish = function( topic, args ) {

        if ( !topics[topic] ) {
            return false;
        }

        var subscribers = topics[topic],
            len = subscribers ? subscribers.length : 0;

        while (len--) {
            subscribers[len].func( topic, args );
        }

        return this;
    };

    // Subscribe to events of interest
    // with a specific topic name and a
    // callback function, to be executed
    // when the topic/event is observed
    q.subscribe = function( topic, func ) {

        if (!topics[topic]) {
            topics[topic] = [];
        }

        var token = ( ++subUid ).toString();
        topics[topic].push({
            token: token,
            func: func
        });
        return token;
    };

    // Unsubscribe from a specific
    // topic, based on a tokenized reference
    // to the subscription
    q.unsubscribe = function( token ) {
        for ( var m in topics ) {
            if ( topics[m] ) {
                for ( var i = 0, j = topics[m].length; i < j; i++ ) {
                    if ( topics[m][i].token === token) {
                        topics[m].splice( i, 1 );
                        return token;
                    }
                }
            }
        }
        return this;
    };
}( pubsub ));
```

Using our implementation

We can now use the implementation to publish and subscribe to events of interest as follows (Example 9-1):

Example 9-1. Using our implementation

```
// Another simple message handler

// A simple message logger that logs any topics and data received through our
// subscriber
var messageLogger = function ( topics, data ) {
    console.log( "Logging: " + topics + ": " + data );
};

// Subscribers listen for topics they have subscribed to and
// invoke a callback function (e.g messageLogger) once a new
// notification is broadcast on that topic
var subscription = pubsub.subscribe( "inbox/newMessage", messageLogger );

// Publishers are in charge of publishing topics or notifications of
// interest to the application. e.g:

pubsub.publish( "inbox/newMessage", "hello world!" );

// or
pubsub.publish( "inbox/newMessage", ["test", "a", "b", "c"] );

// or
pubsub.publish( "inbox/newMessage", {
  sender: "hello@google.com",
  body: "Hey again!"
});

// We can also unsubscribe if we no longer wish for our subscribers
// to be notified
// pubsub.unsubscribe( subscription );

// Once unsubscribed, this for example won't result in our
// messageLogger being executed as the subscriber is
// no longer listening
pubsub.publish( "inbox/newMessage", "Hello! are you still there?" );
```

User-interface notifications

Next, let's imagine we have a web application responsible for displaying real-time stock information.

The application might have a grid for displaying the stock stats and a counter for displaying the last point of update. When the data model changes, the application will need to update the grid and counter. In this scenario, our subject (which will be publishing topics/notifications) is the data model, and our subscribers are the grid and counter.

When our subscribers receive notification that the model itself has changed, they can update accordingly.

In our implementation, our subscriber will listen to the topic `newDataAvailable` to find out if new stock information is available. If a new notification is published to this topic, it will trigger `gridUpdate` to add a new row to our grid containing this information. It will also update a *last updated* counter to log the last time data was added (Example 9-2).

Example 9-2. User-interface notifications

```
// Return the current local time to be used in our UI later
getCurrentTime = function (){

   var date = new Date(),
         m = date.getMonth() + 1,
         d = date.getDate(),
         y = date.getFullYear(),
         t = date.toLocaleTimeString().toLowerCase();

        return (m + "/" + d + "/" + y + " " + t);
};

// Add a new row of data to our fictional grid component
function addGridRow( data ) {

   // ui.grid.addRow( data );
   console.log( "updated grid component with:" + data );

}

// Update our fictional grid to show the time it was last
// updated
function updateCounter( data ) {

   // ui.grid.updateLastChanged( getCurrentTime() );
   console.log( "data last updated at: " + getCurrentTime() + " with " + data);

}
```

```
// Update the grid using the data passed to our subscribers
gridUpdate = function( topic, data ){

  if ( data !== "undefined" ) {
     addGridRow( data );
     updateCounter( data );
   }

};

// Create a subscription to the newDataAvailable topic
var subscriber = pubsub.subscribe( "newDataAvailable", gridUpdate );

// The following represents updates to our data layer. This could be
// powered by ajax requests which broadcast that new data is available
// to the rest of the application.

// Publish changes to the gridUpdated topic representing new entries
pubsub.publish( "newDataAvailable", {
  summary: "Apple made $5 billion",
  identifier: "APPL",
  stockPrice: 570.91
});

pubsub.publish( "newDataAvailable", {
  summary: "Microsoft made $20 million",
  identifier: "MSFT",
  stockPrice: 30.85
});
```

Decoupling applications using Ben Alman's Pub/Sub implementation

In the following movie ratings example, we'll be using Ben Alman's jQuery implementation of Publish/Subscribe to demonstrate how we can decouple a user interface. Notice how submitting a rating only has the effect of publishing the fact that new user and rating data is available.

It's left up to the subscribers to those topics to then delegate what happens with that data. In our case, we're pushing that new data into existing arrays and then rendering them using the Underscore library's `.template()` method for templating.

Here is the HTML/Templates code (Example 9-3):

Example 9-3. HTML/Templates code for Pub/Sub

```
<script id="userTemplate" type="text/html">
   <li><%= name %></li>
</script>


<script id="ratingsTemplate" type="text/html">
   <li><strong><%= title %></strong> was rated <%= rating %>/5</li>
</script>
```

```
<div id="container">

   <div class="sampleForm">
       <p>
           <label for="twitter_handle">Twitter handle:</label>
           <input type="text" id="twitter_handle" />
       </p>
       <p>
           <label for="movie_seen">Name a movie you've seen this year:</label>
           <input type="text" id="movie_seen" />
       </p>
       <p>

           <label for="movie_rating">Rate the movie you saw:</label>
           <select id="movie_rating">
                 <option value="1">1</option>
                  <option value="2">2</option>
                  <option value="3">3</option>
                  <option value="4">4</option>
                  <option value="5" selected>5</option>

          </select>
        </p>
        <p>

            <button id="add">Submit rating</button>
        </p>
    </div>



    <div class="summaryTable">
        <div id="users"><h3>Recent users</h3></div>
        <div id="ratings"><h3>Recent movies rated</h3></div>
    </div>

 </div>
```

Here is the JavaScript code (Example 9-4):

Example 9-4. JavaScript code for Pub/Sub

```
;(function( $ ) {

  // Pre-compile templates and "cache" them using closure
  var
    userTemplate = _.template($( "#userTemplate" ).html()),
    ratingsTemplate = _.template($( "#ratingsTemplate" ).html());

  // Subscribe to the new user topic, which adds a user
  // to a list of users who have submitted reviews
  $.subscribe( "/new/user", function( e, data ){

    var compiledTemplate;
    if( data ){

      compiledTemplate = _.template($( "#userTemplate" ).html());

      $('#users').append( compiledTemplate( data ));

    }

  });


  // Subscribe to the new rating topic. This is composed of a title and
  // rating. New ratings are appended to a running list of added user
  // ratings.
  $.subscribe( "/new/rating", function( e, data ){

    var compiledTemplate;

    if( data ){

      compiledTemplate = _.template($( "#ratingsTemplate" ).html());

      $( "#ratings" ).append( compiledTemplate( data ) );

    }

  });

  // Handler for adding a new user
  $("#add").on("click", function( e ) {

    e.preventDefault();

    var strUser = $("#twitter_handle").val(),
       strMovie = $("#movie_seen").val(),
       strRating = $("#movie_rating").val();

    // Inform the application a new user is available
    $.publish( "/new/user",  { name: strUser } );

    // Inform the app a new rating is available
    $.publish( "/new/rating",  { title: strMovie, rating: strRating} );

    });

})( jQuery );
```

Decoupling an Ajax-based jQuery application

In our final example, we're going to take a practical look at how decoupling our code using Pub/Sub early on in the development process can save us some potentially painful refactoring later on.

Quite often in Ajax-heavy applications, once we've received a response to a request, we want to achieve more than just one unique action. We could simply add all of the post-request logic into a success callback, but there are drawbacks to this approach.

Highly coupled applications sometimes increase the effort required to reuse functionality due to the increased inter-function/code dependency. What this means is that, although keeping our post-request logic hardcoded in a callback might be fine if we're just trying to grab a result set once, it's not as appropriate when we want to make further Ajax calls to the same data source (and different end behavior) without rewriting parts of the code multiple times. Rather than having to go back through each layer that calls the same data source and generalize them later on, we can use pub/sub from the start and save time.

Using Observers, we can also easily separate application-wide notifications regarding different events down to whatever level of granularity we're comfortable with—something that can be less elegantly done using other patterns.

Notice how in our sample below, one topic notification is made when a user indicates that he wants to make a search query and another is made when the request returns and actual data is available for consumption. It's left up to the subscribers to then decide how to use knowledge of these events (or the data returned). The benefits of this are that, if we wanted, we could have 10 different subscribers using the data returned in different ways, but as far as the Ajax-layer is concerned, it doesn't care. Its sole duty is to request and return data, then pass it on to whoever wants to use it. This separation of concerns can make the overall design of our code a little cleaner.

Here is the HTML/Templates code (Example 9-5):

Example 9-5. HTML/Templates code for Ajax

```
<form id="flickrSearch">

   <input type="text" name="tag" id="query"/>

   <input type="submit" name="submit" value="submit"/>

</form>



<div id="lastQuery"></div>

<div id="searchResults"></div>



<script id="resultTemplate" type="text/html">
    <% _.each(items, function( item ){  %>
            <li><p><img src="<%= item.media.m %>"/></p></li>
    <% });%>
</script>
```

Here is the JavaScript code (Example 9-6):

Example 9-6. JavaScript code for Ajax

```
;(function( $ ) {

    // Pre-compile template and "cache" it using closure
    var resultTemplate = _.template($( "#resultTemplate" ).html());

   // Subscribe to the new search tags topic
   $.subscribe( "/search/tags" , function( tags ) {
       $( "#searchResults" )
                .html("Searched for:" + tags + "");
   });

   // Subscribe to the new results topic
   $.subscribe( "/search/resultSet" , function( results ){

       $( "#searchResults" ).append(resultTemplate( results ));

       $( "#searchResults" ).append(compiled_template( results ));

   });

   // Submit a search query and publish tags on the /search/tags topic
   $( "#flickrSearch" ).submit( function( e ) {

       e.preventDefault();
       var tags = $(this).find( "#query").val();

       if ( !tags ){
        return;
       }

       $.publish( "/search/tags" , [ $.trim(tags) ]);

   });


   // Subscribe to new tags being published and perform
   // a search query using them. Once data has returned
   // publish this data for the rest of the application
   // to consume

   $.subscribe("/search/tags", function( tags ) {

       $.getJSON( "http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=?" ,{
              tags: tags,
              tagmode: "any",
              format: "json"
            },
            function( data ){

                if( !data.items.length ) {
                  return;
                }

                $.publish( "/search/resultSet" , data.items  );
        });

    });


})();
```

The Observer pattern is useful for decoupling a number of different scenarios in application design, and if you haven't been using it, I recommend picking up one of the prewritten implementations mentioned here and giving it a try. It's one of the easier design patterns to get started with but also one of the most powerful.

The Mediator Pattern

The dictionary refers to a mediator as "a neutral party that assists in negotiations and conflict resolution."[2] In our world, a mediator is a behavioral design pattern that allows us to expose a unified interface through which the different parts of a system may communicate.

If it appears a system has too many direct relationships between components, it may be time to have a central point of control that components communicate through instead. The Mediator pattern promotes loose coupling by ensuring that instead of components referring to each other explicitly, their interaction is handled through this central point. This can help us decouple systems and improve the potential for component reusability.

A real-world analogy could be a typical airport traffic control system. A tower (mediator) handles which planes can take off and land, because all communications (notifications being listened for or broadcast) are performed from the planes to the control tower, rather than from plane to plane. A centralized controller is key to the success of this system, and that's really the role that the Mediator plays in software design (Figure 9-5).

2. Wikipedia; Dictionary.com

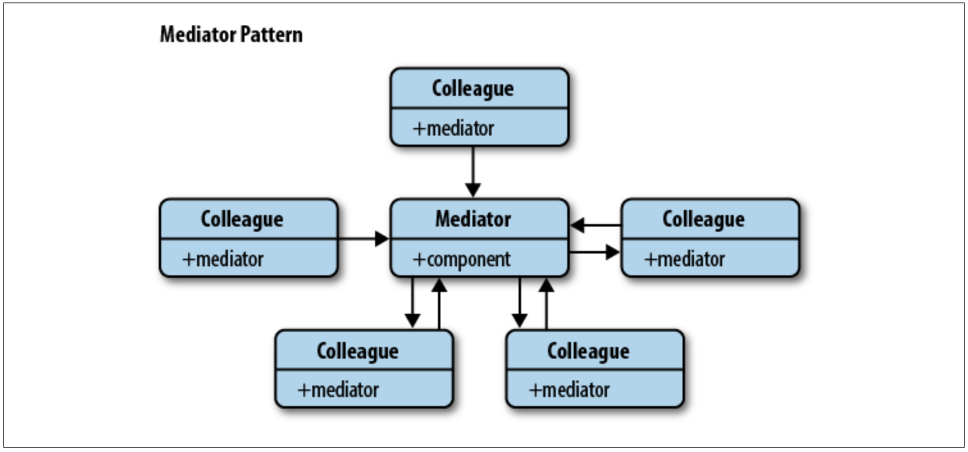


Figure 9-5. Mediator pattern

In implementation terms, the Mediator pattern is essentially a shared subject in the Observer pattern. This might assume that a direction Publish/Subscribe relationship between objects or modules in such systems is sacrificed in order to maintain a central point of contact.

It may also be considered supplemental—perhaps used for application-level notifications such as a communication between different subsystems that are themselves complex and may desire internal component decoupling through Publish/Subscribe relationships.

Another analogy would be DOM event bubbling and event delegation. If all subscriptions in a system are made against the document rather than individual nodes, the document effectively serves as a mediator. Instead of binding to the events of the individual nodes, a higher level object is given the responsibility of notifying subscribers about interaction events.

Basic Implementation

A simple implementation of the Mediator pattern can be found below, exposing both `publish()` and `subscribe()` methods for use:

```
var mediator = (function(){

    // Storage for topics that can be broadcast or listened to
    var topics = {};

    // Subscribe to a topic, supply a callback to be executed
    // when that topic is broadcast to
    var subscribe = function( topic, fn ){

        if ( !topics[topic] ){
          topics[topic] = [];
```

```
            }

            topics[topic].push( { context: this, callback: fn } );

            return this;
        };

        // Publish/broadcast an event to the rest of the application
        var publish = function( topic ){

            var args;

            if ( !topics[topic] ){
              return false;
            }

            args = Array.prototype.slice.call( arguments, 1 );
            for ( var i = 0, l = topics[topic].length; i < l; i++ ) {

                var subscription = topics[topic][i];
                subscription.callback.apply( subscription.context, args );
            }
            return this;
        };

        return {
            publish: publish,
            subscribe: subscribe,
            installTo: function( obj ){
                obj.subscribe = subscribe;
                obj.publish = publish;
            }
        };

    }());
```

Advanced Implementation

For those interested in a more advanced implementation, read on for a walkthrough of my trimmed-down version of Jack Lawson's excellent Mediator.js. Among other improvements, this version supports topic namespaces, subscriber removal, and a much more robust Publish/Subscribe system for our Mediator. If however, you wish to skip this walkthrough, you can go directly to the next example to continue reading.[3]

To start, let's implement the notion of a subscriber, which we can consider an instance of a Mediator's topic registration.

By generating object instances, we can easily update subscribers later without the need to unregister and re-register them. Subscribers can be written as constructors that take a function `fn` to be called, an `options` object, and a `context`.

3. Thanks to Jack for his excellent code comments which assisted with this section.

```
// Pass in a context to attach our Mediator to.
// By default this will be the window object
(function( root ){

  function guidGenerator() { /*..*/}

  // Our Subscriber constructor
  function Subscriber( fn, options, context ){

    if ( !this instanceof Subscriber ) {

      return new Subscriber( fn, context, options );

    }else{

      // guidGenerator() is a function that generates
      // GUIDs for instances of our Mediators Subscribers so
      // we can easily reference them later on. We're going
      // to skip its implementation for brevity

      this.id = guidGenerator();
      this.fn = fn;
      this.options = options;
      this.context = context;
      this.topic = null;

    }
  }
})();
```

Topics in our Mediator hold a list of callbacks and subtopics that are fired when `Mediator.Publish` is called on our Mediator instance. It also contains methods for manipulating lists of data.

```
// Let's model the Topic.
// JavaScript lets us use a Function object as a
// conjunction of a prototype for use with the new
// object and a constructor function to be invoked.
function Topic( namespace ){

  if ( !this instanceof Topic ) {
    return new Topic( namespace );
  }else{

    this.namespace = namespace || "";
    this._callbacks = [];
    this._topics = [];
    this.stopped = false;

  }
}


// Define the prototype for our topic, including ways to
// add new subscribers or retrieve existing ones.
```

```
Topic.prototype = {

  // Add a new subscriber
  AddSubscriber: function( fn, options, context ){

    var callback = new Subscriber( fn, options, context );

    this._callbacks.push( callback );

    callback.topic = this;

    return callback;
  },
...
```

Our topic instance is passed along as an argument to the Mediator callback. Further callback propagation can then be called using a handy utility method called `StopPropagation()`:

```
StopPropagation: function(){
  this.stopped = true;
},
```

We can also make it easy to retrieve existing subscribers when given a GUID identifier:

```
GetSubscriber: function( identifier ){

  for(var x = 0, y = this._callbacks.length; x < y; x++ ){
    if( this._callbacks[x].id == identifier || this._callbacks[x].fn == identifier ){
      return this._callbacks[x];
    }
  }

  for( var z in this._topics ){
    if( this._topics.hasOwnProperty( z ) ){
      var sub = this._topics[z].GetSubscriber( identifier );
      if( sub !== undefined ){
        return sub;
      }
    }
  }

},
```

Next, in case we need them, we can offer easy ways to add new topics, check for existing topics, or retrieve topics:

```
AddTopic: function( topic ){
  this._topics[topic] = new Topic( (this.namespace ? this.namespace + ":" : "") + topic );
},

HasTopic: function( topic ){
  return this._topics.hasOwnProperty( topic );
},

ReturnTopic: function( topic ){
  return this._topics[topic];
},
```

We can explicitly remove subscribers if we feel they are no longer necessary. The following will recursively remove a Subscriber through its subtopics:

```
    RemoveSubscriber: function( identifier ){

      if( !identifier ){
        this._callbacks = [];

        for( var z in this._topics ){
          if( this._topics.hasOwnProperty(z) ){
            this._topics[z].RemoveSubscriber( identifier );
          }
        }
      }

      for( var y = 0, x = this._callbacks.length; y < x; y++ ) {
        if( this._callbacks[y].fn == identifier || this._callbacks[y].id == identifier ){
          this._callbacks[y].topic = null;
          this._callbacks.splice( y,1 );
          x--; y--;
        }
      }

    },
```

Next, we include the ability to `Publish` arbitrary arguments to subscribers recursively through subtopics.

```
    Publish: function( data ){

      for( var y = 0, x = this._callbacks.length; y < x; y++ ) {

          var callback = this._callbacks[y], l;
            callback.fn.apply( callback.context, data );

        l = this._callbacks.length;

        if( l < x ){
          y--;
          x = l;
        }
      }

      for( var x in this._topics ){
        if( !this.stopped ){
          if( this._topics.hasOwnProperty( x ) ){
            this._topics[x].Publish( data );
          }
        }
      }

      this.stopped = false;
    }
  };
```

Here we expose the `Mediator` instance we will primarily be interacting with. It is through here that events are registered and removed from topics.

```
  function Mediator() {

    if ( !this instanceof Mediator ) {
      return new Mediator();
    }else{
      this._topics = new Topic( "" );
    }

  };
```

For more advanced-use cases, we can get our Mediator supporting namespaces for topics such as `inbox:messages:new:read.GetTopic`, which in the following example, returns topic instances based on a namespace.

```
  Mediator.prototype = {

    GetTopic: function( namespace ){
      var topic = this._topics,
          namespaceHierarchy = namespace.split( ":" );

      if( namespace === "" ){
        return topic;
      }

      if( namespaceHierarchy.length > 0 ){
        for( var i = 0, j = namespaceHierarchy.length; i < j; i++ ){

          if( !topic.HasTopic( namespaceHierarchy[i]) ){
            topic.AddTopic( namespaceHierarchy[i] );
          }

          topic = topic.ReturnTopic( namespaceHierarchy[i] );
        }
      }

      return topic;
    },
```

In this section, we define a `Mediator.Subscribe` method, which accepts a topic namespace, a function `fn` to be executed, `options`, and once again a `context` to call the function into `Subscribe`. This creates a topic if one doesn't exist.

```
    Subscribe: function( topiclName, fn, options, context ){
      var options = options || {},
          context = context || {},
          topic = this.GetTopic( topicName ),
          sub = topic.AddSubscriber( fn, options, context );

      return sub;
    },
```

Continuing on from this, we can define further utilities for accessing specific subscribers or removing them from topics recursively.

```
        // Returns a subscriber for a given subscriber id / named function and topic namespace

        GetSubscriber: function( identifier, topic ){
          return this.GetTopic( topic || "" ).GetSubscriber( identifier );
        },

        // Remove a subscriber from a given topic namespace recursively based on
        // a provided subscriber id or named function.

        Remove: function( topicName, identifier ){
          this.GetTopic( topicName ).RemoveSubscriber( identifier );
        },
```

Our primary `Publish` method allows us to arbitrarily publish data to a chosen topic namespace.

Topics are called recursively downward. For example, a post to `inbox:messages` will post to `inbox:messages:new` and `inbox:messages:new:read`. It is used as follows:

```
        Mediator.Publish( "inbox:messages:new", [args] );
        Publish: function( topicName ){
          var args = Array.prototype.slice.call( arguments, 1),
              topic = this.GetTopic( topicName );

          args.push( topic );

          this.GetTopic( topicName ).Publish( args );
        }
      };
```

Finally we can easily expose our Mediator for attachment to the object passed in to `root`:

```
      root.Mediator = Mediator;
      Mediator.Topic = Topic;
      Mediator.Subscriber = Subscriber;

    // Remember we can pass anything in here. I've passed in window to
    // attach the Mediator to, but we can just as easily attach it to another
    // object if desired.
    })( window );
```

Example

Using either of the implementations from above (both the simple option and the more advanced one), we can then put together a simple chat logging system as follows.

Here is the HTML code:

```
     <h1>Chat</h1>
    <form id="chatForm">
        <label for="fromBox">Your Name:</label>
        <input id="fromBox" type="text"/>
        <br />
        <label for="toBox">Send to:</label>
        <input id="toBox" type="text"/>
        <br />
        <label for="chatBox">Message:</label>
        <input id="chatBox" type="text"/>
        <button type="submit">Chat</button>
    </form>

    <div id="chatResult"></div>
```

Here is the JavaScript code:

```
$( "#chatForm" ).on( "submit", function(e) {
    e.preventDefault();

    // Collect the details of the chat from our UI
    var text = $( "#chatBox" ).val(),
        from = $( "#fromBox" ).val(),
        to = $( "#toBox" ).val();

    // Publish data from the chat to the newMessage topic
    mediator.publish( "newMessage" , { message: text, from: from, to: to } );
});

// Append new messages as they come through
function displayChat( data ) {
    var date = new Date(),
        msg = data.from + " said \"" + data.message + "\" to " + data.to;

    $( "#chatResult" )
        .prepend("" + msg + " (" + date.toLocaleTimeString() + ")");
}

// Log messages
function logChat( data ) {
    if ( window.console ) {
        console.log( data );
    }
}



// Subscribe to new chat messages being submitted
// via the mediator
mediator.subscribe( "newMessage", displayChat );
mediator.subscribe( "newMessage", logChat );


// The following will however only work with the more advanced implementation:

function amITalkingToMyself( data ) {
    return data.from === data.to;
}

function iAmClearlyCrazy( data ) {
    $( "#chatResult" ).prepend("" + data.from + " is talking to himself.");
}

 mediator.Subscribe( amITalkingToMyself, iAmClearlyCrazy );
```

Advantages and Disadvantages

The largest benefit of the Mediator pattern is that it reduces the communication channels needed between objects or components in a system from many to many to just many to one. Adding new publishers and subscribers is relatively easy due to the level of decoupling present.

Perhaps the biggest downside of using the pattern is that it can introduce a single point of failure. Placing a Mediator between modules can cause a performance hit as they are always communicating indirectly. Because of the nature of loose coupling, it's difficult to establish how a system might react by only looking at the broadcasts.

That said, it's useful to remind ourselves that decoupled systems have a number of other benefits: if our modules communicated with each other directly, changes to modules (e.g., another module throwing an exception) could easily have a domino effect on the rest of our application. This problem is less of a concern with decoupled systems.

At the end of the day, tight coupling causes all kinds of headaches, and this is just another alternative solution, but one that can work very well if implemented correctly.

Mediator Versus Observer

Developers often wonder what the differences are between the Mediator pattern and the Observer pattern. Admittedly, there is a bit of overlap, but let's refer back to the GoF for an explanation:

> In the Observer pattern, there is no single object that encapsulates a constraint. Instead, the Observer and the Subject must cooperate to maintain the constraint. Communication patterns are determined by the way observers and subjects are interconnected: a single subject usually has many observers, and sometimes the observer of one subject is a subject of another observer.

Both Mediators and Observers promote loose coupling; however, the Mediator pattern achieves this by having objects communicate strictly through the Mediator. The Observer pattern creates observable objects that publish events of interest to objects that are subscribed to them.

Mediator Versus Facade

We will be covering the Facade pattern shortly, but for reference purposes, some developers may also wonder whether there are similarities between the Mediator and

Facade patterns. They do both abstract the functionality of existing modules, but there are some subtle differences.

The Mediator pattern centralizes communication between modules where it's explicitly referenced by these modules. In a sense, this is multidirectional. The Facade pattern, on the other hand, just defines a simpler interface to a module or system but doesn't add any additional functionality. Other modules in the system aren't directly aware of the concept of a facade and could be considered unidirectional.

The Prototype Pattern

The GoF refers to the Prototype pattern as one that creates objects based on a template of an existing object through cloning.

We can think of the Prototype pattern as being based on prototypal inheritance in which we create objects that act as prototypes for other objects. The `prototype` object itself is effectively used as a blueprint for each object the constructor creates. If the prototype of the constructor function used contains a property called `name` for example (as per the code sample that follows), then each object created by that same constructor will also have this same property (Figure 9-6).

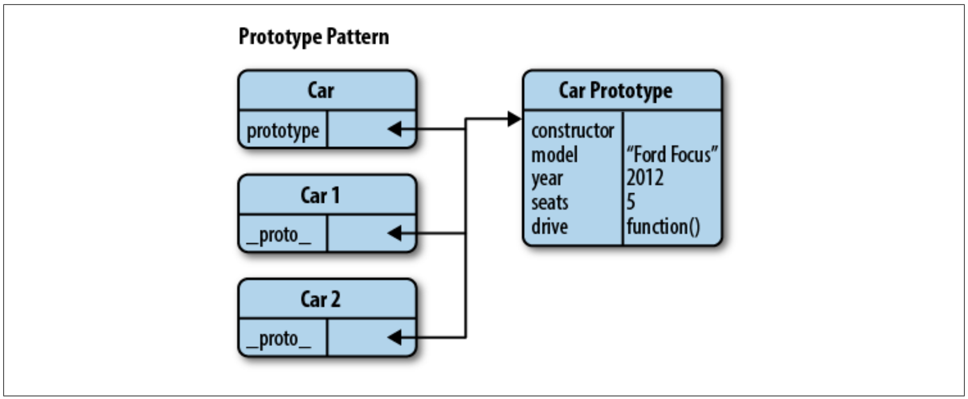


Figure 9-6. Prototype pattern

Reviewing the definitions for this pattern in existing (non-JavaScript) literature, we *may* find references to classes once again. The reality is that prototypal inheritance avoids using classes altogether. There isn't a "definition" object nor a core object in theory; we're simply creating copies of existing functional objects.

One of the benefits of using the Prototype pattern is that we're working with the prototypal strengths JavaScript has to offer natively rather than attempting to imitate features of other languages. With other design patterns, this isn't always the case.

Not only is the pattern an easy way to implement inheritance, but it can come with a performance boost as well: when defining functions in an object, they're all created by reference (so all child objects point to the same function), instead of creating their own individual copies.

For those interested, real prototypal inheritance, as defined in the ECMAScript 5 standard, requires the use of `Object.create` (which we previously looked at earlier in this section). To review, `Object.create` creates an object that has a specified prototype and optionally contains specified properties as well (e.g., `Object.create( prototype, optionalDescriptorObjects )`).

We can see this demonstrated in the following example:

```
var myCar = {

  name: "Ford Escort",

  drive: function () {
    console.log( "Weeee. I'm driving!" );
  },

  panic: function () {
    console.log( "Wait. How do you stop this thing?" );
  }

};

// Use Object.create to instantiate a new car
var yourCar = Object.create( myCar );

// Now we can see that one is a prototype of the other
console.log( yourCar.name );
```

`Object.create` also allows us to easily implement advanced concepts such as differential inheritance in which objects are able to directly inherit from other objects. We saw earlier that `Object.create` allows us to initialize object properties using the second supplied argument. For example:

```
var vehicle = {
  getModel: function () {
    console.log( "The model of this vehicle is.." + this.model );
  }
};

var car = Object.create(vehicle, {

  "id": {
    value: MY_GLOBAL.nextId(),
    // writable:false, configurable:false by default
    enumerable: true
  },

  "model": {
    value: "Ford",
    enumerable: true
  }
});
```

Here, the properties can be initialized on the second argument of `Object.create` using an object literal with a syntax similar to that used by the `Object.defineProperties` and `Object.defineProperty` methods that we looked at previously.

It is worth noting that prototypal relationships can cause trouble when enumerating properties of objects and (as Crockford recommends) wrapping the contents of the loop in a `hasOwnProperty()` check.

If we wish to implement the Prototype pattern without directly using `Object.create`, we can simulate the pattern as per the above example as follows:

```
var vehiclePrototype = {

  init: function ( carModel ) {
    this.model = carModel;
  },

  getModel: function () {
    console.log( "The model of this vehicle is.." + this.model);
  }
};


function vehicle( model ) {

  function F() {};
  F.prototype = vehiclePrototype;

  var f = new F();

  f.init( model );
  return f;

}

var car = vehicle( "Ford Escort" );
car.getModel();
```

This alternative does not allow the user to define read-only properties in the same manner (as the `vehiclePrototype` may be altered if not careful).

A final alternative implementation of the Prototype pattern could be the following:

```
var beget = (function () {

    function F() {}

    return function ( proto ) {
        F.prototype = proto;
```

```
            return new F();
        };
    })();
```

One could reference this method from the `vehicle` function. Note, however that `vehicle` here is emulating a constructor, since the Prototype pattern does not include any notion of initialization beyond linking an object to a prototype.

The Command Pattern

The Command pattern aims to encapsulate method invocation, requests, or operations into a single object and gives us the ability to both parameterize and pass method calls around that can be executed at our discretion. In addition, it enables us to decouple objects invoking the action from the objects that implement them, giving us a greater degree of overall flexibility in swapping out concrete *classes* (objects).

Concrete classes are best explained in terms of class-based programming languages and are related to the idea of abstract classes. An *abstract* class defines an interface, but doesn't necessarily provide implementations for all of its member functions. It acts as a base class from which others are derived. A derived class that implements the missing functionality is called a *concrete* class (Figure 9-7).

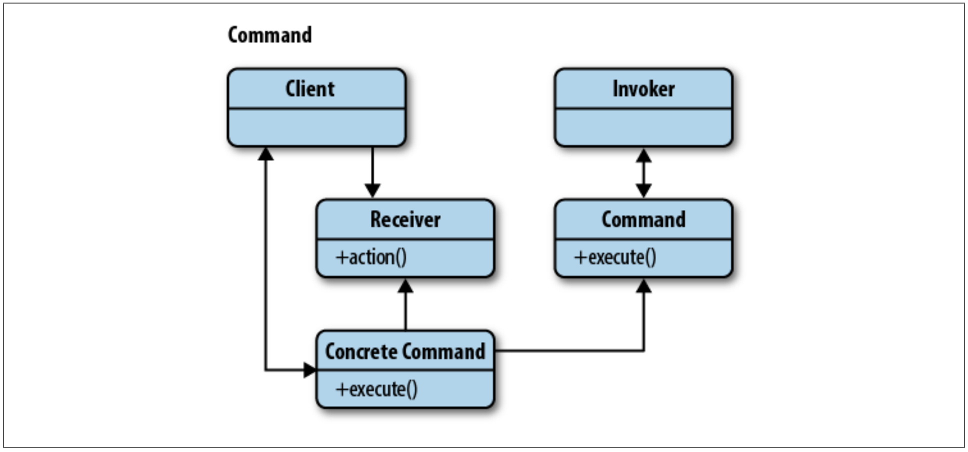


Figure 9-7. Command pattern

The general idea behind the Command pattern is that it provides us a means to separate the responsibilities of issuing commands from anything executing commands, delegating this responsibility to different objects instead.

Implementation wise, simple command objects bind together both an action and the object wishing to invoke the action. They consistently include an execution operation (such as `run()` or `execute()`). All Command objects with the same interface can easily be swapped as needed, and this is considered one of the larger benefits of the pattern.

To demonstrate the Command pattern we're going to create a simple car purchasing service.

```
(function(){

  var CarManager = {

      // request information
      requestInfo: function( model, id ){
        return "The information for " + model + " with ID " + id + " is foobar";
      },

      // purchase the car
      buyVehicle: function( model, id ){
        return "You have successfully purchased Item " + id + ", a " + model;
      },

      // arrange a viewing
      arrangeViewing: function( model, id ){
        return "You have successfully booked a viewing of " + model + " ( " + id + " ) ";
      }

    };

})();
```

Taking a look at the above code, it would be trivial to invoke our `CarManager` methods by directly accessing the object. We would all be forgiven for thinking there is nothing wrong with this—technically, it's completely valid JavaScript. There are however scenarios in which this may be disadvantageous.

For example, imagine if the core API behind the `CarManager` changed. This would require all objects directly accessing these methods within our application to also be modified. This could be viewed as a layer of coupling, which effectively goes against the OOP methodology of loosely coupling objects as much as possible. Instead, we could solve this problem by abstracting the API away further.

Let's now expand on our `CarManager` so that our application of the Command pattern results in the following: accept any named methods that can be performed on the `CarManager` object, passing along any data that might be used, such as the Car model and ID.

Here is what we would like to be able to achieve:

```
CarManager.execute( "buyVehicle", "Ford Escort", "453543" );
```

As per this structure we should now add a definition for the `CarManager.execute` method as follows:

```
CarManager.execute = function ( name ) {
    return CarManager[name] && CarManager[name].apply( CarManager, [].slice.call(arguments, 1) );
};
```

Our final sample calls would thus look as follows:

```
    CarManager.execute( "arrangeViewing", "Ferrari", "14523" );
    CarManager.execute( "requestInfo", "Ford Mondeo", "54323" );
    CarManager.execute( "requestInfo", "Ford Escort", "34232" );
    CarManager.execute( "buyVehicle", "Ford Escort", "34232" );
```

The Facade Pattern

When we put up a facade, we present an outward appearance to the world that may conceal a very different reality. This was the inspiration for the name behind the next pattern we're going to review: the Facade pattern. This pattern provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers, something that almost always improves usability (Figure 9-8).

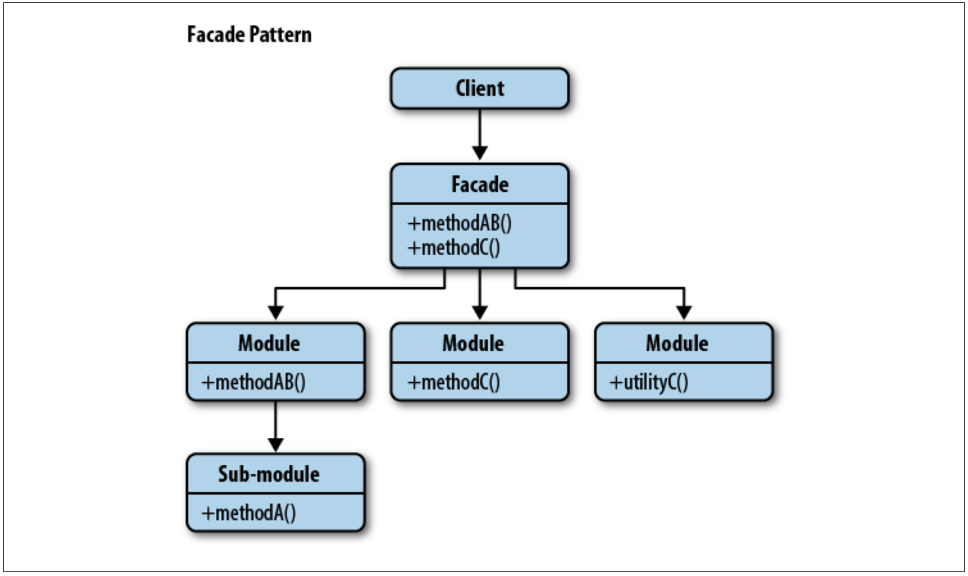


Figure 9-8. Facade pattern

Facades are a structural pattern that can often be seen in JavaScript libraries such as jQuery where, although an implementation may support methods with a wide range of behaviors, only a "facade," or limited abstraction of these methods, is presented to the public for use.

This allows us to interact with the Facade directly rather than the subsystem behind the scenes. Whenever we use jQuery's `$(el).css()` or `$(el).animate()` methods, we're actually using a Facade: the simpler public interface that lets us avoid having to manually call the many internal methods in jQuery core required to get some behavior working. This also circumvents the need to manually interact with DOM APIs and maintain state variables.

The jQuery core methods should be considered intermediate abstractions. The more immediate burden to developers is the DOM API, and facades are what make the jQuery library so easy to use.

To build on what we've learned, the Facade pattern both simplifies the interface of a class and decouples the class from the code that uses it. This gives us the ability to indirectly interact with subsystems in a way that can sometimes be less prone to error than accessing the subsystem directly. A Facade's advantages include ease of use and often a small-sized footprint in implementing the pattern.

Let's take a look at the pattern in action. This is an unoptimized code example, but here we're using a Facade to simplify an interface for listening to events across browsers. We do this by creating a common method that can be used in one's code which does the task of checking for the existence of features so that it can provide a safe and cross-browser compatible solution.

```
var addMyEvent = function( el,ev,fn ){

   if( el.addEventListener ){
            el.addEventListener( ev,fn, false );
      }else if(el.attachEvent){
            el.attachEvent( "on" + ev, fn );
      } else{
           el["on" + ev] = fn;
    }

};
```

In a similar manner, we're all familiar with jQuery's `$(document).ready(..)`. Internally, this is actually powered by a method called `bindReady()`, which is doing this:

```
bindReady: function() {
    ...
    if ( document.addEventListener ) {
      // Use the handy event callback
      document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );

      // A fallback to window.onload, that will always work
      window.addEventListener( "load", jQuery.ready, false );

    // If IE event model is used
    } else if ( document.attachEvent ) {

      document.attachEvent( "onreadystatechange", DOMContentLoaded );

      // A fallback to window.onload, that will always work
      window.attachEvent( "onload", jQuery.ready );
               ...
```

This is another example of a Facade, in which the rest of the world simply uses the limited interface exposed by `$(document).ready(..)` and the more complex implementation powering it is kept hidden from sight.

Facades don't just have to be used on their own, however. They can also be integrated with other patterns, such as the Module pattern. As we can see below, our instance of the Module pattern contains a number of methods that have been privately defined. A Facade is then used to supply a much simpler API to accessing these methods:

```
var module = (function() {

    var _private = {
        i:5,
        get : function() {
            console.log( "current value:" + this.i);
        },
        set : function( val ) {
            this.i = val;
        },
        run : function() {
            console.log( "running" );
        },
        jump: function(){
            console.log( "jumping" );
        }
    };

    return {

        facade : function( args ) {
            _private.set(args.val);
            _private.get();
            if ( args.run ) {
                _private.run();
            }
        }
    };
}());


// Outputs: "current value: 10" and "running"
module.facade( {run: true, val:10} );
```

In this example, calling `module.facade()` will actually trigger a set of private behavior within the module, but again, the users aren't concerned with this. We've made it much easier for them to consume a feature without needing to worry about implementation-level details.

Notes on Abstraction

Facades generally have few disadvantages, but one concern worth noting is performance. Namely, one must determine whether there is an implicit cost to the abstraction a Facade offers to our implementation and, if so, whether this cost is justifiable. Going back to the jQuery library, most of us are aware that both `getElementById("identifier")` and `$("#identifier")` can be used to query an element on a page by its ID.

Did you know however that `getElementById()` on its own is significantly faster by a high order of magnitude? Take a look at this jsPerf test to see results on a per-browser level: *http://jsperf.com/getelementbyid-vs-jquery-id*. Now of course, we have to keep in mind that jQuery (and Sizzle, its selector engine) are doing a lot more behind the scenes to optimize our query (and that a jQuery object, not just a DOM node is returned).

The challenge with this particular Facade is that in order to provide an elegant selector function capable of accepting and parsing multiple types of queries, there is an implicit cost of abstraction. The user isn't required to access `jQuery.getById("identifier")` or `jQuery.getbyClass("identifier")` and so on. That said, the trade-off in performance has been tested in practice over the years and, given the success of jQuery, a simple Facade actually worked out very well for the team.

When using the Facade pattern, try to be aware of any performance costs involved and make a call on whether they are worth the level of abstraction offered.

The Factory Pattern

The Factory pattern is another creational pattern concerned with the notion of creating objects. Where it differs from the other patterns in its category is that it doesn't explicitly require the use of a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created (Figure 9-9).

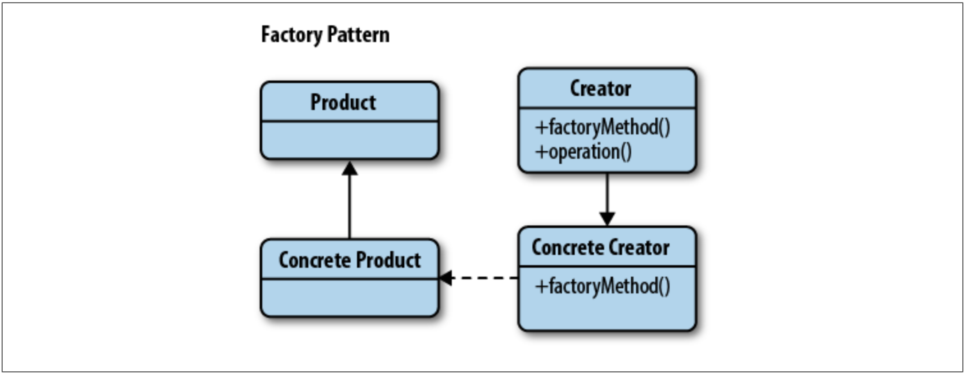


Figure 9-9. Factory pattern

Imagine that we have a UI factory where we are asked to create a type of UI component. Rather than creating this component directly using the `new` operator or via another creational constructor, we ask a Factory object for a new component instead. We inform the Factory what type of object is required (e.g., "Button", "Panel") and it instantiates this, returning it to us for use.

This is particularly useful if the object creation process is relatively complex—e.g., if it strongly depends on dynamic factors or application configuration.

Examples of this pattern can be found in UI libraries such as ExtJS, where the methods for creating objects or components may be further subclassed.

The following is an example that builds upon our previous snippets using the Constructor pattern logic to define cars. It demonstrates how a `vehicle` factory may be implemented using the Factory pattern:

```
// Types.js - Constructors used behind the scenes

// A constructor for defining new cars
function Car( options ) {

  // some defaults
  this.doors = options.doors || 4;
  this.state = options.state || "brand new";
  this.color = options.color || "silver";

}

// A constructor for defining new trucks
function Truck( options){

  this.state = options.state || "used";
  this.wheelSize = options.wheelSize || "large";
  this.color = options.color || "blue";
}


// FactoryExample.js

// Define a skeleton vehicle factory
function VehicleFactory() {}

// Define the prototypes and utilities for this factory

// Our default vehicleClass is Car
VehicleFactory.prototype.vehicleClass = Car;

// Our Factory method for creating new Vehicle instances
VehicleFactory.prototype.createVehicle = function ( options ) {

  if( options.vehicleType === "car" ){
    this.vehicleClass = Car;
  }else{
    this.vehicleClass = Truck;
  }

  return new this.vehicleClass( options );

};

// Create an instance of our factory that makes cars
var carFactory = new VehicleFactory();
var car = carFactory.createVehicle( {
            vehicleType: "car",
                color: "yellow",
                doors: 6 } );

// Test to confirm our car was created using the vehicleClass/prototype Car

// Outputs: true
console.log( car instanceof Car );

// Outputs: Car object of color "yellow", doors: 6 in a "brand new" state
console.log( car );
```

In Approach 1, we modify a `VehicleFactory` instance to use the `Truck` class:

```
var movingTruck = carFactory.createVehicle( {
                      vehicleType: "truck",
                      state: "like new",
                      color: "red",
                      wheelSize: "small" } );

// Test to confirm our truck was created with the vehicleClass/prototype Truck

// Outputs: true
console.log( movingTruck instanceof Truck );

// Outputs: Truck object of color "red", a "like new" state
// and a "small" wheelSize
console.log( movingTruck );
```

In Approach 2, we subclass `VehicleFactory` to create a factory class that builds `Trucks`:

```
function TruckFactory () {}
TruckFactory.prototype = new VehicleFactory();
TruckFactory.prototype.vehicleClass = Truck;

var truckFactory = new TruckFactory();
var myBigTruck = truckFactory.createVehicle( {
                    state: "omg..so bad.",
                    color: "pink",
                    wheelSize: "so big" } );

// Confirms that myBigTruck was created with the prototype Truck
// Outputs: true
console.log( myBigTruck instanceof Truck );

// Outputs: Truck object with the color "pink", wheelSize "so big"
// and state "omg. so bad"
console.log( myBigTruck );
```

When to Use the Factory Pattern

The Factory pattern can be especially useful when applied to the following situations:

- When our object or component setup involves a high level of complexity.
- When we need to easily generate different instances of objects depending on the environment we are in.

- When we're working with many small objects or components that share the same properties.
- When composing objects with instances of other objects that need only satisfy an API contract (a.k.a., duck typing) to work. This is useful for decoupling.

When Not to Use the Factory Pattern

When applied to the wrong type of problem, this pattern can introduce an unnecessarily great deal of complexity to an application. Unless providing an interface for object creation is a design goal for the library or framework we are writing, I would suggest sticking to explicit constructors to avoid the unnecessary overhead.

Due to the fact that the process of object creation is effectively abstracted behind an interface, this can also introduce problems with unit testing depending on just how complex this process might be.

Abstract Factories

It is also useful to be aware of the Abstract Factory pattern, which aims to encapsulate a group of individual factories with a common goal. It separates the details of implementation of a set of objects from their general usage.

An Abstract Factory pattern should be used where a system must be independent from the way the objects it creates are generated or it needs to work with multiple types of objects.

An example that is both simple and easier to understand is a vehicle factory, which defines ways to get or register vehicles types. The abstract factory can be named `AbstractVehicleFactory`. The abstract factory will allow the definition of types of vehicle like `car` or `truck`, and concrete factories will implement only classes that fulfill the vehicle contract (e.g., `Vehicle.prototype.drive` and `Vehicle.prototype.breakDown`).

```
var AbstractVehicleFactory = (function () {

  // Storage for our vehicle types
  var types = {};

  return {
      getVehicle: function ( type, customizations ) {
          var Vehicle = types[type];

          return (Vehicle ? new Vehicle(customizations) : null);
      },

      registerVehicle: function ( type, Vehicle ) {
          var proto = Vehicle.prototype;

          // only register classes that fulfill the vehicle contract
          if ( proto.drive && proto.breakDown ) {
                types[type] = Vehicle;
            }

            return AbstractVehicleFactory;
        }
    };
})();


// Usage:

AbstractVehicleFactory.registerVehicle( "car", Car );
AbstractVehicleFactory.registerVehicle( "truck", Truck );

// Instantiate a new car based on the abstract vehicle type
var car = AbstractVehicleFactory.getVehicle( "car" , {
            color: "lime green",
            state: "like new" } );

// Instantiate a new truck in a similar manner
var truck = AbstractVehicleFactory.getVehicle( "truck" , {
            wheelSize: "medium",
            color: "neon yellow" } );
```

The Mixin Pattern

In traditional programming languages such as C++ and Lisp, Mixins are classes that offer functionality that can be easily inherited by a sub-class or group of sub-classes for the purpose of function reuse.

Subclassing

For developers unfamiliar with subclassing, we will go through a brief beginners' primer before diving into Mixins and Decorators further.

Subclassing is a term that refers to inheriting properties for a new object from a base or *superclass* object. In traditional object-oriented programming, a class `B` is able to extend another class `A`. Here we consider `A` a superclass and `B` a subclass of `A`. As such, all instances of `B` inherit the methods from `A`. `B` is however still able to define its own methods, including those that override methods originally defined by `A`.

Should `B` need to invoke a method in `A` that has been overridden, we refer to this as method chaining. Should `B` need to invoke the constructor `A` (the superclass), we call this constructor chaining.

To demonstrate subclassing, we first need a base object that can have new instances of itself created. Let's model this around the concept of a person.

```
var Person =  function( firstName , lastName ){

  this.firstName = firstName;
```

```
      this.lastName =  lastName;
      this.gender = "male";

    };
```

Next, we'll want to specify a new class (object) that's a subclass of the existing Person object. Let us imagine we want to add distinct properties to distinguish a Person from a Superhero while inheriting the properties of the Person superclass. As superheroes share many common traits with normal people (e.g. name, gender), this should hopefully illustrate how subclassing works adequately.

```
    // a new instance of Person can then easily be created as follows:
    var clark = new Person( "Clark" , "Kent" );

    // Define a subclass constructor for for "Superhero":
    var Superhero = function( firstName, lastName , powers ){

        // Invoke the superclass constructor on the new object
        // then use .call() to invoke the constructor as a method of
        // the object to be initialized.

        Person.call( this, firstName, lastName );

        // Finally, store their powers, a new array of traits not found in a normal "Person"
        this.powers = powers;
    };

    SuperHero.prototype = Object.create( Person.prototype );
    var superman = new Superhero( "Clark" ,"Kent" , ["flight","heat-vision"] );
    console.log( superman );

    // Outputs Person attributes as well as powers
```

The Superhero constructor creates an object that descends from Person. Objects of this type have attributes of the objects that are above them in the chain, and if we had set default values in the Person object, Superhero is capable of overriding any inherited values with values specific to its object.

Mixins

In JavaScript, we can look at inheriting from Mixins as a means of collecting functionality through extension. Each new object we define has a prototype from which it can inherit further properties. Prototypes can inherit from other object prototypes but, even more importantly, can define properties for any number of object instances. We can leverage this fact to promote function reuse (Figure 9-10).

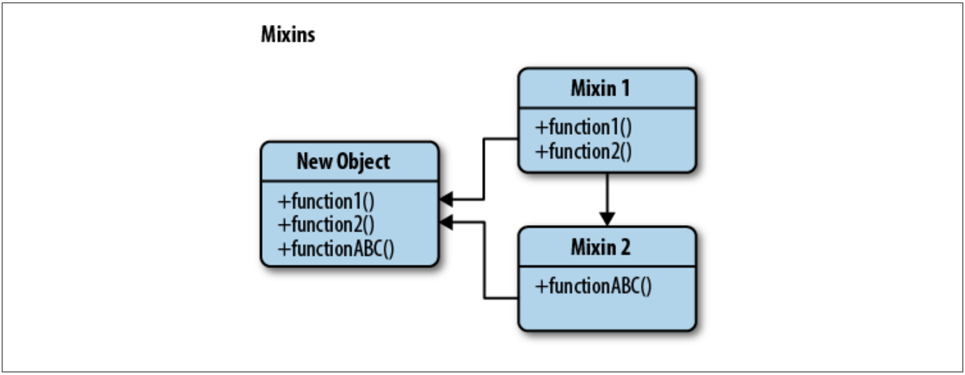


Figure 9-10. Mixins

Mixins allow objects to borrow (or inherit) functionality from them with a minimal amount of complexity. As the pattern works well with JavaScript's object prototypes, it gives us a fairly flexible way to share functionality from not just one Mixin, but effectively many through multiple inheritance.

They can be viewed as objects with attributes and methods that can be easily shared across a number of other object prototypes. Imagine that we define a Mixin containing utility functions in a standard object literal as follows:

```
var myMixins = {

  moveUp: function(){
    console.log( "move up" );
  },

  moveDown: function(){
    console.log( "move down" );
  },

  stop: function(){
    console.log( "stop! in the name of love!" );
  }

};
```

We can then easily extend the prototype of existing constructor functions to include this behavior using a helper such as the Underscore.js `_.extend()` method:

```
// A skeleton carAnimator constructor
function carAnimator(){
  this.moveLeft = function(){
    console.log( "move left" );
  };
}

// A skeleton personAnimator constructor
function personAnimator(){
  this.moveRandomly = function(){ /*..*/ };
}

// Extend both constructors with our Mixin
_.extend( carAnimator.prototype, myMixins );
_.extend( personAnimator.prototype, myMixins );

// Create a new instance of carAnimator
var myAnimator = new carAnimator();
myAnimator.moveLeft();
myAnimator.moveDown();
myAnimator.stop();

// Outputs:
// move left
// move down
// stop! in the name of love!
```

As we can see, this allows us to easily "mix" in common behaviour into object constructors fairly trivially.

In the next example, we have two constructors: a `Car` and a `Mixin`. What we're going to do is augment (another way of saying extend) the `Car` so that it can inherit specific methods defined in the `Mixin`, namely `driveForward()` and `driveBackward()`. This time, we won't be using Underscore.js.

Instead, this example will demonstrate how to augment a constructor to include functionality without the need to duplicate this process for every constructor function we may have.

```
// Define a simple Car constructor
var Car = function ( settings ) {

        this.model = settings.model || "no model provided";
        this.color = settings.color || "no colour provided";

    };

// Mixin
var Mixin = function () {};

Mixin.prototype = {

    driveForward: function () {
        console.log( "drive forward" );
    },

    driveBackward: function () {
        console.log( "drive backward" );
    },

    driveSideways: function () {
        console.log( "drive sideways" );
    }
```

```
};


// Extend an existing object with a method from another
function augment( receivingClass, givingClass ) {

    // only provide certain methods
    if ( arguments[2] ) {
        for ( var i = 2, len = arguments.length; i < len; i++ ) {
            receivingClass.prototype[arguments[i]] = givingClass.prototype[arguments[i]];
        }
    }
    // provide all methods
    else {
        for ( var methodName in givingClass.prototype ) {

            // check to make sure the receiving class doesn't
            // have a method of the same name as the one currently
            // being processed
            if ( !Object.hasOwnProperty(receivingClass.prototype, methodName) ) {
                receivingClass.prototype[methodName] = givingClass.prototype[methodName];
            }

            // Alternatively:
            // if ( !receivingClass.prototype[methodName] ) {
            //  receivingClass.prototype[methodName] = givingClass.prototype[methodName];
            // }
        }
    }
}


// Augment the Car constructor to include "driveForward" and "driveBackward"
augment( Car, Mixin, "driveForward", "driveBackward" );

// Create a new Car
var myCar = new Car({
    model: "Ford Escort",
    color: "blue"
});

// Test to make sure we now have access to the methods
myCar.driveForward();
myCar.driveBackward();

// Outputs:
// drive forward
// drive backward

// We can also augment Car to include all functions from our mixin
// by not explicitly listing a selection of them
augment( Car, Mixin );

var mySportsCar = new Car({
    model: "Porsche",
    color: "red"
});

mySportsCar.driveSideways();

// Outputs:
// drive sideways
```

Advantages and Disadvantages

Mixins assist in decreasing functional repetition and increasing function reuse in a system. Where an application is likely to require shared behavior across object instances, we can easily avoid any duplication by maintaining this shared functionality in a Mixin and thus focusing on implementing only the functionality in our system which is truly distinct.

That said, the downsides to Mixins are a little more debatable. Some developers feel that injecting functionality into an object prototype is a bad idea as it leads to both prototype pollution and a level of uncertainly regarding the origin of our functions. In large systems, this may well be the case.

I would argue that strong documentation can assist in minimizing the amount of confusion regarding the source of mixed-in functions, but as with every pattern, if care is taken during implementation, we should be okay.

The Decorator Pattern

Decorators are a structural design pattern that aim to promote code reuse. Similar to Mixins, they can be considered another viable alternative to object subclassing.

Classically, Decorators offered the ability to add behavior to existing classes in a system dynamically. The idea was that the *decoration* itself wasn't essential to the base functionality of the class; otherwise, it would be baked into the *superclass* itself.

They can be used to modify existing systems where we wish to add additional features to objects without the need to heavily modify the underlying code using them. A common reason why developers use them is that their applications may contain features requiring a large quantity of distinct types of object. Imagine having to define hundreds of different object constructors for, say, a JavaScript game (Figure 9-11).

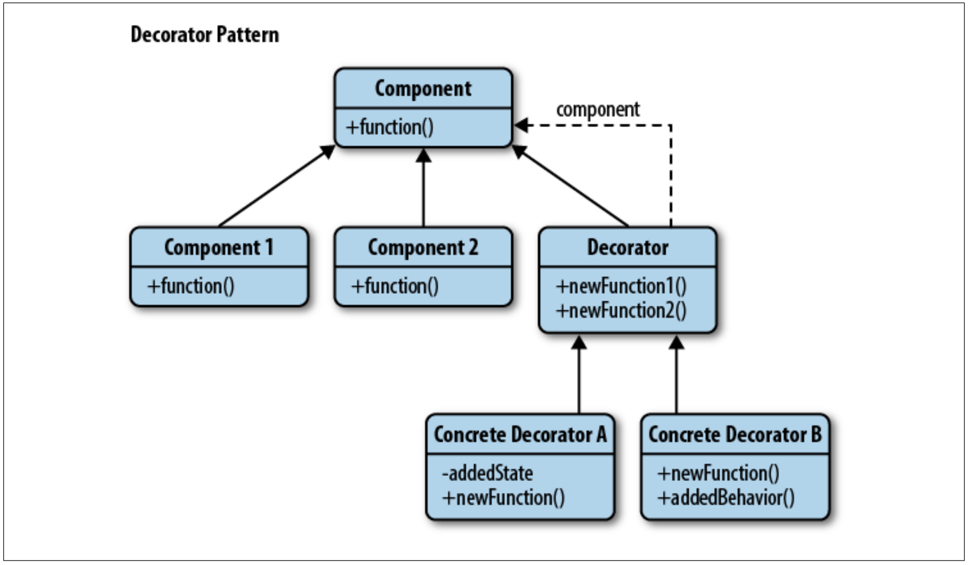


Figure 9-11. Decorator pattern

The object constructors could represent distinct player types, each with differing capabilities. A *Lord of the Rings* game could require constructors for `Hobbit`, `Elf`, `Orc`, `Wizard`, `Mountain Giant`, `Stone Giant`, and so on, but there could easily be hundreds of these. If we then factored in capabilities, imagine having to create subclasses for each combination of capability type—e.g., `HobbitWithRing`, `HobbitWithSword`, `HobbitWithRingAndSword`, and so on. This isn't very practical and certainly isn't manageable when we factor in a growing number of different abilities.

The Decorator pattern isn't heavily tied to how objects are created but instead focuses on the problem of extending their functionality. Rather than just relying on prototypal inheritance, we work with a single base object and progressively add decorator objects that provide the additional capabilities. The idea is that rather than subclassing, we add (decorate) properties or methods to a base object so it's a little more streamlined.

Adding new attributes to objects in JavaScript is a very straightforward process, so with this in mind, a very simplistic decorator may be implemented as follows (Examples 9-7 and 9-8):

Example 9-7. Decorating Constructors with New Functionality

```
// A vehicle constructor
function vehicle( vehicleType ){

    // some sane defaults
    this.vehicleType = vehicleType || "car";
    this.model = "default";
    this.license = "00000-000";
}

// Test instance for a basic vehicle
var testInstance = new vehicle( "car" );
console.log( testInstance );

// Outputs:
// vehicle: car, model:default, license: 00000-000

// Lets create a new instance of vehicle, to be decorated
var truck = new vehicle( "truck" );

// New functionality we're decorating vehicle with
truck.setModel = function( modelName ){
    this.model = modelName;
};

truck.setColor = function( color ){
    this.color = color;
};

// Test the value setters and value assignment works correctly
truck.setModel( "CAT" );
truck.setColor( "blue" );

console.log( truck );

// Outputs:
// vehicle:truck, model:CAT, color: blue

// Demonstrate "vehicle" is still unaltered
var secondInstance = new vehicle( "car" );
console.log( secondInstance );

// Outputs:
// vehicle: car, model:default, license: 00000-000
```

This type of simplistic implementation is functional, but it doesn't really demonstrate all of the strengths Decorators have to offer. For this, we're first going to go through my variation of the Coffee example from an excellent book called *Head First Design Patterns* by Freeman, Sierra and Bates, which is modeled around a Macbook purchase.

Example 9-8. Decorating Objects with Multiple Decorators

```
// The constructor to decorate
function MacBook() {

  this.cost = function () { return 997; };
  this.screenSize = function () { return 11.6; };

}

// Decorator 1
function Memory( macbook ) {
  var v = macbook.cost();
  macbook.cost = function() {
    return v + 75;
  };

}

// Decorator 2
function Engraving( macbook ){

  var v = macbook.cost();
  macbook.cost = function(){
    return  v + 200;
  };

}

// Decorator 3
function Insurance( macbook ){

  var v = macbook.cost();
  macbook.cost = function(){
     return  v + 250;
  };

}

var mb = new MacBook();
Memory( mb );
Engraving( mb );
Insurance( mb );

// Outputs: 1522
console.log( mb.cost() );

// Outputs: 11.6
console.log( mb.screenSize() );
```

In the example, our Decorators are overriding the `MacBook()` superclass object's `.cost()` function to return the current price of the `Macbook` plus the cost of the upgrade being specified.

It's considered a decoration' as the original `Macbook` objects' constructor methods that are not overridden (e.g. `screenSize()`), as well as any other properties that we may define as a part of the `Macbook`, remain unchanged and intact.

There isn't really a defined *interface* in the previous example, and we're shifting away the responsibility of ensuring an object meets an interface when moving from the creator to the receiver.

Pseudoclassical Decorators

We're now going to examine a variation of the Decorator first presented in a JavaScript form in *Pro JavaScript Design Patterns* (PJDP) by Dustin Diaz and Ross Harmes.

Unlike some of the examples from earlier, Diaz and Harmes stick more closely to how decorators are implemented in other programming languages (such as Java or C++) using the concept of an "interface," which we will define in more detail shortly.

This particular variation of the Decorator pattern is provided for reference purposes. If you find it overly complex, I recommend opting for one of the simpler implementations covered earlier.

Interfaces

PJDP describes the Decorator pattern as one that is used to transparently wrap objects inside other objects of the same interface. An interface is a way of defining the methods an object *should* have; however, it doesn't actually directly specify how those methods should be implemented.

Interfaces can also indicate what parameters the methods take, but this is considered optional.

So, why would we use an interface in JavaScript? The idea is that they're self-documenting and promote reusability. In theory, interfaces also make code more stable by ensuring changes to them must also be made to the objects implementing them.

Below is an example of an implementation of interfaces in JavaScript using duck-typing, an approach that helps determine whether an object is an instance of that constructor/object based on the methods it implements.

```
// Create interfaces using a pre-defined Interface
// constructor that accepts an interface name and
// skeleton methods to expose.

// In our reminder example summary() and placeOrder()
// represent functionality the interface should
// support
var reminder = new Interface( "List", ["summary", "placeOrder"] );

var properties = {
  name: "Remember to buy the milk",
  date: "05/06/2016",
  actions:{
    summary: function (){
      return "Remember to buy the milk, we are almost out!";
   },
    placeOrder: function (){
      return "Ordering milk from your local grocery store";
    }
```

```
      }
    };

    // Now create a constructor implementing the above properties
    // and methods

    function Todo( config ){

      // State the methods we expect to be supported
      // as well as the Interface instance being checked
      // against

      Interface.ensureImplements( config.actions, reminder );

      this.name = config.name;
      this.methods = config.actions;

    }

    // Create a new instance of our Todo constructor

    var todoItem = Todo( properties );

    // Finally test to make sure these function correctly

    console.log( todoItem.methods.summary() );
    console.log( todoItem.methods.placeOrder() );

    // Outputs:
    // Remember to buy the milk, we are almost out!
    // Ordering milk from your local grocery store
```

In this example, `Interface.ensureImplements` provides strict functionality checking, and code for both this and the `Interface` constructor can be found here.

The biggest problem with interfaces is that, as there isn't built-in support for them in JavaScript, there is a danger of attempting to emulate a feature of another language that may not be an ideal fit. Lightweight interfaces can be used without a great performance cost, however, and we will next look at *Abstract* Decorators using this same concept.

Abstract Decorators

To demonstrate the structure of this version of the Decorator pattern, we're going to imagine we have a superclass that models a `Macbook` once again and a store that allows us to "decorate" our Macbook with a number of enhancements for an additional fee.

Enhancements can include upgrades to 4 GB or 8 GB of RAM, engraving, Parallels, or a case. Now if we were to model this using an individual subclass for each combination of enhancement options, it might look something like this:

```
    var Macbook = function(){
            //...
    };
```

```
var  MacbookWith4GBRam =  function(){},
     MacbookWith8GBRam = function(){},
     MacbookWith4GBRamAndEngraving = function(){},
     MacbookWith8GBRamAndEngraving = function(){},
     MacbookWith8GBRamAndParallels = function(){},
     MacbookWith4GBRamAndParallels = function(){},
     MacbookWith8GBRamAndParallelsAndCase = function(){},
     MacbookWith4GBRamAndParallelsAndCase = function(){},
     MacbookWith8GBRamAndParallelsAndCaseAndInsurance = function(){},
     MacbookWith4GBRamAndParallelsAndCaseAndInsurance = function(){};
```

... and so on.

This would be an impractical solution, as a new subclass would be required for every possible combination of enhancements that are available. As we would prefer to keep things simple without maintaining a large set of subclasses, let's look at how decorators may be used to solve this problem better.

Rather than requiring all of the combinations we saw earlier, we should simply have to create five new decorator classes. Methods that are called on these enhancement classes would be passed on to our `Macbook` class.

In our next example, decorators transparently wrap around their components and can interestingly be interchanged as they use the same interface.

Here's the interface we're going to define for the Macbook:

```
var Macbook = new Interface( "Macbook",
  ["addEngraving",
  "addParallels",
  "add4GBRam",
  "add8GBRam",
  "addCase"]);

// A Macbook Pro might thus be represented as follows:
var MacbookPro = function(){
    // implements Macbook
};

MacbookPro.prototype = {
    addEngraving: function(){
    },
    addParallels: function(){
    },
    add4GBRam: function(){
    },
    add8GBRam:function(){
    },
    addCase: function(){
    },
    getPrice: function(){
      // Base price
      return 900.00;
```

```
        }
    };
```

To make it easier for us to add as many more options as needed later on, an Abstract Decorator class is defined with default methods required to implement the `Macbook` interface, which the rest of the options will subclass. Abstract Decorators ensure that we can decorate a base class independently with as many decorators as needed in different combinations (remember the example earlier?) without needing to derive a class for every possible combination.

```
    // Macbook decorator abstract decorator class

    var MacbookDecorator = function( macbook ){

        Interface.ensureImplements( macbook, Macbook );
        this.macbook = macbook;

    };

    MacbookDecorator.prototype = {
        addEngraving: function(){
            return this.macbook.addEngraving();
        },
        addParallels: function(){
            return this.macbook.addParallels();
        },
        add4GBRam: function(){
            return this.macbook.add4GBRam();
        },
        add8GBRam:function(){
            return this.macbook.add8GBRam();
        },
        addCase: function(){
            return this.macbook.addCase();
        },
        getPrice: function(){
            return this.macbook.getPrice();
        }
    };
```

What's happening in the above sample is that the `Macbook` Decorator is taking an object to use as the component. It's using the `Macbook` interface we defined earlier, and for each method, it calls the same method on the component. We can now create our option classes just by using the `Macbook` Decorator; simply call the superclass constructor, and any methods can be overridden as necessary.

```
    var CaseDecorator = function( macbook ){

       // call the superclass's constructor next
       this.superclass.constructor( macbook );

    };

    // Let's now extend the superclass
```

```
extend( CaseDecorator, MacbookDecorator );

CaseDecorator.prototype.addCase = function(){
    return this.macbook.addCase() + "Adding case to macbook";
};

CaseDecorator.prototype.getPrice = function(){
    return this.macbook.getPrice() + 45.00;
};
```

As we can see, most of this is relatively straightforward to implement. What we're doing is overriding the `addCase()` and `getPrice()` methods that need to be decorated, and we're achieving this by first executing the component's method and then adding to it.

As there's been quite a lot of information presented in this section so far; let's try to bring it all together in a single example that will hopefully highlight what we have learned.

```
// Instantiation of the macbook
var myMacbookPro = new MacbookPro();

// Outputs: 900.00
console.log( myMacbookPro.getPrice() );

// Decorate the macbook
myMacbookPro = new CaseDecorator( myMacbookPro );

// This will return 945.00
console.log( myMacbookPro.getPrice() );
```

As decorators are able to modify objects dynamically, they're a perfect pattern for changing existing systems. Occasionally, it's just simpler to create decorators around an object versus the trouble of maintaining individual subclasses for each object type. This makes maintaining applications that may require a large number of subclassed objects significantly more straightforward.

Decorators with jQuery

As with other patterns we've covered, there are also examples of the Decorator pattern that can be implemented with jQuery. `jQuery.extend()` allows us to extend (or merge) two or more objects (and their properties) together into a single object either at runtime or dynamically at a later point.

In this scenario, a target object can be decorated with new functionality without necessarily breaking or overriding existing methods in the source/superclass object (although this can be done).

In the following example, we define three objects: `defaults`, `options`, and `settings`. The aim of the task is to decorate the `defaults` object with additional functionality found in `optionssettings`. We must first leave `defaults` in an untouched state where we don't lose the ability to access the properties or functions found in it at a later point; then, gain the ability to use the decorated properties and functions found in `options`:

```
var decoratorApp = decoratorApp || {};

// define the objects we're going to use
decoratorApp = {

    defaults: {
        validate: false,
        limit: 5,
        name: "foo",
        welcome: function () {
            console.log( "welcome!" );
        }
    },

    options: {
        validate: true,
        name: "bar",
        helloWorld: function () {
            console.log( "hello world" );
        }
    },

    settings: {},

    printObj: function ( obj ) {
        var arr = [],
            next;
        $.each( obj, function ( key, val ) {
            next = key + ": ";
            next += $.isPlainObject(val) ? printObj( val ) : val;
            arr.push( next );
        } );

        return "{ " + arr.join(", ") + " }";
    }

};

// merge defaults and options, without modifying defaults explicitly
decoratorApp.settings = $.extend({}, decoratorApp.defaults, decoratorApp.options);

// what we have done here is decorated defaults in a way that provides
// access to the properties and functionality it has to offer (as well as
// that of the decorator "options"). defaults itself is left unchanged

$("#log")
    .append( decoratorApp.printObj(decoratorApp.settings) +
          + decoratorApp.printObj(decoratorApp.options) +
          + decoratorApp.printObj(decoratorApp.defaults));

// settings -- { validate: true, limit: 5, name: bar,
welcome: function (){ console.log( "welcome!" ); },
// helloWorld: function (){ console.log("hello!"); } }
// options -- { validate: true, name: bar,
helloWorld: function (){ console.log("hello!"); } }
// defaults -- { validate: false, limit: 5, name: foo,
welcome: function (){ console.log("welcome!"); } }
```

Advantages and Disadvantages

Developers enjoy using this pattern, as it can be used transparently and is also fairly flexible: as we've seen, objects can be wrapped or "decorated" with new behavior and then continue to be used without needing to worry about the base object being modified. In a broader context, this pattern also allows us to avoid relying on large numbers of subclasses to get the same benefits.

There are however drawbacks that we should be aware of when implementing the pattern. If poorly managed, it can significantly complicate our application architecture, as it introduces many small but similar objects into our namespace. The concern here is that, in addition to becoming hard to manage, other developers unfamiliar with the pattern may have a hard time grasping why it's being used.

Sufficient commenting or pattern research should assist with the latter; however, as long as we keep a handle on how widespread we use the decorator in our applications, we should be fine on both counts.

Flyweight

The Flyweight pattern is a classical structural solution for optimizing code that is repetitive, slow, and inefficiently shares data. It aims to minimize the use of memory in an application by sharing as much data as possible with related objects (e.g., application configuration, state, and so on—see Figure 9-12).

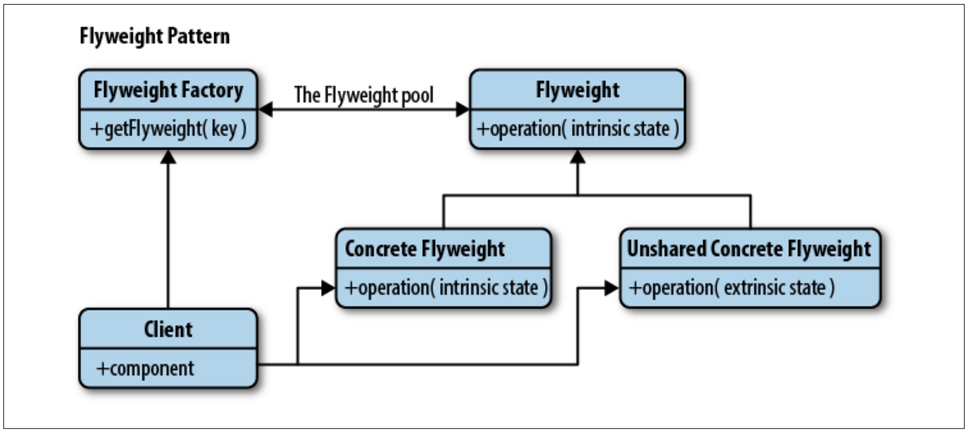


Figure 9-12. Flyweight pattern

The pattern was first conceived by Paul Calder and Mark Linton in 1990 and was named after the boxing weight class that includes fighters weighing less than 112 lb. The name *Flyweight* itself is derived from this weight classification, as it refers to the small weight (memory footprint) the pattern aims to help us achieve.

In practice, Flyweight data sharing can involve taking several similar objects or data constructs used by a number of objects and placing this data into a single external object. We can pass through this object through to those depending on this data, rather than storing identical data across each one.

Using Flyweights

There are two ways in which the Flyweight pattern can be applied. The first is at the data layer, where we deal with the concept of sharing data between large quantities of similar objects stored in memory.

The second is at the DOM layer, where the Flyweight can be used as a central event-manager to avoid attaching event handlers to every child element in a parent container we wish to have some similar behavior.

As the data layer is where the Flyweight pattern is most used traditionally, we'll take a look at this first.

Flyweights and Sharing Data

For this application, there are a few more concepts around the classical Flyweight pattern that we need to be aware of. In the Flyweight pattern, there's a concept of two states: intrinsic and extrinsic. Intrinsic information may be required by internal methods in our objects, which they absolutely cannot function without. Extrinsic information can however be removed and stored externally.

Objects with the same intrinsic data can be replaced with a single shared object, created by a factory method. This allows us to reduce the overall quantity of implicit data being stored quite significantly.

The benefit of this is that we're able to keep an eye on objects that have already been instantiated so that new copies are only ever created should the intrinsic state differ from the object we already have.

We use a manager to handle the extrinsic states. How this is implemented can vary, but one approach is to have the manager object contain a central database of the extrinsic states and the flyweight objects to which they belong.

Implementing Classical Flyweights

As the Flyweight pattern hasn't been heavily used in JavaScript in recent years, many of the implementations we might use for inspiration come from the Java and C++ worlds.

Our first look at Flyweights in code is my JavaScript implementation of the Java sample of the Flyweight pattern from Wikipedia (*http://en.wikipedia.org/wiki/Flyweight_pattern*).

We will be making use of three types of Flyweight components in this implementation, which are listed below:

Flyweight
: Corresponds to an interface through which flyweights are able to receive and act on extrinsic states.

Concrete flyweight
: Actually implements the Flyweight interface and stores intrinsic states. Concrete Flyweights need to be sharable and capable of manipulating a state that is extrinsic.

Flyweight factory
: Manages flyweight objects and creates them, too. It makes sure that our flyweights are shared and manages them as a group of objects that can be queried if we require individual instances. If an object has already been created in the group, it returns it. Otherwise, it adds a new object to the pool and returns it.

These correspond to the following definitions in our implementation:

- `CoffeeOrder`: Flyweight
- `CoffeeFlavor`: Concrete flyweight
- `CoffeeOrderContext`: Helper
- `CoffeeFlavorFactory`: Flyweight factory
- `testFlyweight`: Utilization of our flyweights

Duck punching "implements"

Duck punching allows us to extend the capabilities of a language or solution without necessarily needing to modify the runtime source. As this next solution requires the use of a Java keyword (`implements`) for implementing interfaces and isn't found in JavaScript natively, let's first duck punch it.

`Function.prototype.implementsFor` works on an object constructor and will accept a parent class (function) or object and either inherit from this using normal inheritance (for functions) or virtual inheritance (for objects).

```
// Simulate pure virtual inheritance/"implement" keyword for JS
Function.prototype.implementsFor = function( parentClassOrObject ){
    if ( parentClassOrObject.constructor === Function )
```

```
        {
            // Normal Inheritance
            this.prototype = new parentClassOrObject();
            this.prototype.constructor = this;
            this.prototype.parent = parentClassOrObject.prototype;
        }
        else
        {
            // Pure Virtual Inheritance
            this.prototype = parentClassOrObject;
            this.prototype.constructor = this;
            this.prototype.parent = parentClassOrObject;
        }
        return this;
    };
```

We can use this to patch the lack of an `implements` keyword by having a function inherit an interface explicitly. Below, `CoffeeFlavor` implements the `CoffeeOrder` interface and must contain its interface methods in order for us to assign the functionality powering these implementations to an object.

```
    // Flyweight object
    var CoffeeOrder = {

      // Interfaces
      serveCoffee:function(context){},
        getFlavor:function(){}

    };


    // ConcreteFlyweight object that creates ConcreteFlyweight
    // Implements CoffeeOrder
    function CoffeeFlavor( newFlavor ){

        var flavor = newFlavor;

        // If an interface has been defined for a feature
        // implement the feature
        if( typeof this.getFlavor === "function" ){
          this.getFlavor = function() {
              return flavor;
          };
        }

        if( typeof this.serveCoffee === "function" ){
          this.serveCoffee = function( context ) {
            console.log("Serving Coffee flavor "
              + flavor
              + " to table number "
              + context.getTable());
        };
        }

    }
```

```
// Implement interface for CoffeeOrder
CoffeeFlavor.implementsFor( CoffeeOrder );


// Handle table numbers for a coffee order
function CoffeeOrderContext( tableNumber ) {
   return{
      getTable: function() {
         return tableNumber;
     }
   };
}



// FlyweightFactory object
function CoffeeFlavorFactory() {
    var flavors = [];

  return {
      getCoffeeFlavor: function( flavorName ) {

          var flavor = flavors[flavorName];
          if (flavor === undefined) {
              flavor = new CoffeeFlavor( flavorName );
              flavors.push( [flavorName, flavor] );
          }
          return flavor;
      },

      getTotalCoffeeFlavorsMade: function() {
        return flavors.length;
      }
  };
}

// Sample usage:
// testFlyweight()

function testFlyweight(){


  // The flavors ordered.
  var flavors = new CoffeeFlavor(),

  // The tables for the orders.
    tables = new CoffeeOrderContext(),

  // Number of orders made
    ordersMade = 0,

  // The CoffeeFlavorFactory instance
    flavorFactory;
  function takeOrders( flavorIn, table) {
     flavors[ordersMade] = flavorFactory.getCoffeeFlavor( flavorIn );
     tables[ordersMade++] = new CoffeeOrderContext( table );
  }

   flavorFactory = new CoffeeFlavorFactory();

   takeOrders("Cappuccino", 2);
   takeOrders("Cappuccino", 2);
   takeOrders("Frappe", 1);
   takeOrders("Frappe", 1);
   takeOrders("Xpresso", 1);
   takeOrders("Frappe", 897);
   takeOrders("Cappuccino", 97);
   takeOrders("Cappuccino", 97);
   takeOrders("Frappe", 3);
   takeOrders("Xpresso", 3);
   takeOrders("Cappuccino", 3);
   takeOrders("Xpresso", 96);
   takeOrders("Frappe", 552);
   takeOrders("Cappuccino", 121);
   takeOrders("Xpresso", 121);

   for (var i = 0; i < ordersMade; ++i) {
       flavors[i].serveCoffee(tables[i]);
   }
   console.log(" ");
   console.log("total CoffeeFlavor objects made: " +  flavorFactory.getTotalCoffeeFlavorsMade());
}
```

Converting Code to Use the Flyweight Pattern

Next, let's continue our look at Flyweights by implementing a system to manage all of the books in a library. The important metadata for each book could probably be broken down as follows:

- ID
- Title
- Author
- Genre
- Page count
- Publisher ID
- ISBN

We'll also require the following properties to keep track of which member has checked out a particular book, the date that patron has checked it out on, as well as the expected date of return.

- `checkoutDate`

- `checkoutMember`
- `dueReturnDate`
- `availability`

Each book would thus be represented as follows, prior to any optimization using the Flyweight pattern:

```
var Book = function( id, title, author, genre, pageCount,publisherID,
ISBN, checkoutDate, checkoutMember, dueReturnDate,availability ){

   this.id = id;
   this.title = title;
   this.author = author;
   this.genre = genre;
   this.pageCount = pageCount;
   this.publisherID = publisherID;
   this.ISBN = ISBN;
   this.checkoutDate = checkoutDate;
   this.checkoutMember = checkoutMember;
   this.dueReturnDate = dueReturnDate;
   this.availability = availability;

};

Book.prototype = {

  getTitle: function () {
     return this.title;
  },

  getAuthor: function () {
     return this.author;
  },

  getISBN: function (){
     return this.ISBN;
  },

  // For brevity, other getters are not shown
  updateCheckoutStatus: function( bookID, newStatus, checkoutDate,
  checkoutMember, newReturnDate ){

     this.id  = bookID;
     this.availability = newStatus;
     this.checkoutDate = checkoutDate;
     this.checkoutMember = checkoutMember;
     this.dueReturnDate = newReturnDate;

  },

  extendCheckoutPeriod: function( bookID, newReturnDate ){

      this.id =  bookID;
      this.dueReturnDate = newReturnDate;
  },

  isPastDue: function(bookID){

     var currentDate = new Date();
     return currentDate.getTime() > Date.parse( this.dueReturnDate );

   }
};
```

This probably works fine initially for small collections of books; however, as the library expands to include a larger inventory with multiple versions and copies of each book available, we may find the management system running slower and slower over time. Using thousands of book objects may overwhelm the available memory, but we can optimize our system using the Flyweight pattern to improve this.

We can now separate our data into intrinsic and extrinsic states as follows: data relevant to the book object (`title`, `author`, etc.) is intrinsic, while the checkout data (`checkoutMember`, `dueReturnDate`, etc.) is considered extrinsic. Effectively, this means that only one `Book` object is required for each combination of book properties. It's still a considerable quantity of objects, but significantly fewer than we had previously.

The following single instance of our book metadata combinations will be shared among all of the copies of a book with a particular title.

```
// Flyweight optimized version
var Book = function ( title, author, genre, pageCount, publisherID, ISBN ) {

    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;

};
```

As we can see, the extrinsic states have been removed. Everything to do with library check-outs will be moved to a manager, and as the object data is now segmented, a factory can be used for instantiation.

A Basic Factory

Let's now define a very basic factory. First, we must perform a check to see if a book with a particular title has been previously created inside the system. If it has, we'll return it; if not, a new book will be created and stored so that it can be accessed later. This makes sure that we only create a single copy of each unique intrinsic piece of data:

```
// Book Factory singleton
var BookFactory = (function () {
  var existingBooks = {}, existingBook;
```

```
    return {
      createBook: function ( title, author, genre, pageCount, publisherID, ISBN ) {

        // Find out if a particular book meta-data combination has been created before
        // !! or (bang bang) forces a boolean to be returned
        existingBook = existingBooks[ISBN];
        if ( !!existingBook ) {
          return existingBook;
        } else {

          // if not, let's create a new instance of the book and store it
          var book = new Book( title, author, genre, pageCount, publisherID, ISBN );
          existingBooks[ISBN] = book;
          return book;

        }
      }
    };

  });
```

Managing the Extrinsic States

Next, we need to store the states that were removed from the `Book` objects somewhere. Luckily, a manager (which we'll be defining as a Singleton) can be used to encapsulate them. Combinations of a `Book` object and the library member that's checked it out will be called book records. Our manager will be storing both and will also include checkout-related logic we stripped out during our flyweight optimization of the `Book` class.

```
  // BookRecordManager singleton
  var BookRecordManager = (function () {

    var bookRecordDatabase = {};

    return {
      // add a new book into the library system
      addBookRecord: function ( id, title, author, genre, pageCount, publisherID, ISBN, checkoutDate,
      checkoutMember, dueReturnDate, availability ) {

        var book = bookFactory.createBook( title, author, genre, pageCount,
        publisherID, ISBN );

        bookRecordDatabase[id] = {
          checkoutMember: checkoutMember,
          checkoutDate: checkoutDate,
          dueReturnDate: dueReturnDate,
          availability: availability,
          book: book
        };
      },
      updateCheckoutStatus: function ( bookID, newStatus, checkoutDate,
      checkoutMember, newReturnDate ) {

        var record = bookRecordDatabase[bookID];
        record.availability = newStatus;
        record.checkoutDate = checkoutDate;
        record.checkoutMember = checkoutMember;
        record.dueReturnDate = newReturnDate;
      },

      extendCheckoutPeriod: function ( bookID, newReturnDate ) {
        bookRecordDatabase[bookID].dueReturnDate = newReturnDate;
      },

      isPastDue: function ( bookID ) {
        var currentDate = new Date();
        return currentDate.getTime() > Date.parse(
        bookRecordDatabase[bookID].dueReturnDate );
      }
    };

  });
```

The result of these changes is that all of the data that's been extracted from the `Book` *class* is now being stored in an attribute of the `BookManager` singleton (`BookDatabase`)—something considerably more efficient than the large number of objects we were previously using. Methods related to book checkouts are also now based here, as they deal with data that's extrinsic rather than intrinsic.

This process does add a little complexity to our final solution; however, it's a small concern when compared to the performance issues that have been tackled. Data-wise, if we have 30 copies of the same book, we are now only storing it once. Also, every function takes up memory. With the Flyweight pattern, these functions exist in one place (on the manager) and not on every object, thus saving on memory use.

The Flyweight Pattern and the DOM

The Document Object Model (DOM) supports two approaches that allow objects to detect events: top down (event capture) and bottom up (event bubbling).

In event capture, the event is first captured by the outermost element and propagated to the innermost element. In event bubbling, the event is captured and given to the innermost element and then propagated to the outer elements.

One of the best metaphors for describing Flyweights in this context was written by Gary Chisholm and it goes a little like this:

> Try to think of the flyweight in terms of a pond. A fish opens its mouth (the event), bubbles rise to the surface (the bubbling), a fly sitting on the top flies away when the bubble reaches the surface (the action). In this example we can easily transpose the fish opening its mouth to a button being clicked, the bubbles as the bubbling effect, and the fly flying away to some function being run.

Bubbling was introduced to handle situations in which a single event (e.g., a click) may be handled by multiple event handlers defined at different levels of the DOM hierarchy. Where this happens, event bubbling executes event handlers defined for specific elements at the lowest level possible. From there on, the event bubbles up to containing elements before going to those even higher up.

Flyweights can be used to tweak the event bubbling process further, as we will see shortly (Example 9-9).

For our first practical example, imagine we have a number of similar elements in a document with similar behavior executed when a user action (e.g., click, mouseover) is performed against them.

Normally what we do when constructing our own accordion component, menu, or other list-based widget is bind a click event to each link element in the parent container (e.g., `$('ul li a').on(..)`). Instead of binding the click to multiple elements, we can easily attach a Flyweight to the top of our container, which can listen for events coming from below. These can then be handled using logic that is as simple or complex as required.

As the types of components mentioned often have the same repeating markup for each section (e.g., each section of an accordion), there's a good chance the behavior of each element that may be clicked is going to be quite similar and relative to similar classes nearby. We'll use this information to construct a very basic accordion using the Flyweight below.

A `stateManager` namespace is used here to encapsulate our flyweight logic while jQuery is used to bind the initial click to a container `div`. To ensure that no other logic on the page is attaching similar handles to the container, an `unbind` event is first applied.

Now to establish exactly what child element in the container is clicked, we make use of a `target` check, which provides a reference to the element that was clicked, regardless of its parent. We then use this information to handle the click event without actually needing to bind the event to specific children when our page loads.

Example 9-9. Centralized event handling

Here is the HTML code:

```
<div id="container">
   <div class="toggle" href="#">More Info (Address)
       <span class="info">
           This is more information
       </span></div>
   <div class="toggle" href="#">Even More Info (Map)
       <span class="info">
          <iframe src="http://www.map-generator.net/extmap.php?name=London&amp;
          address=london%2C%20england&amp;width=500...gt;"</iframe>
       </span>
   </div>
</div>
```

Here is the JavaScript code:

```
var stateManager = {

  fly: function () {

    var self = this;

    $( "#container" ).unbind().on( "click" , function ( e ) {
      var target = $( e.originalTarget || e.srcElement );
        if ( target.is( "div.toggle") ) {
          self.handleClick( target );
        }
    });
  },

  handleClick: function ( elem ) {
    elem.find( "span" ).toggle( "slow" );
  }
};
```

The benefit here is that we're converting many independent actions into a shared one (potentially saving on memory).

In our second example, we'll reference some further performance gains that can be achieved using Flyweights with jQuery.

James Padolsey previously wrote an article called *76 bytes for faster jQuery* where he reminded us that each time jQuery fires off a callback, regardless of type (filter, each, event handler), we're able to access the function's context (the DOM element related to it) via the `this` keyword.

Unfortunately, many of us have become used to the idea of wrapping `this` in `$()` or `jQuery()`, which means that a new instance of jQuery is unnecessarily constructed every time, rather than simply doing this (Example 9-10):

Example 9-10. Using the Flyweight for performance optimization

```
$("div").on( "click", function () {
  console.log( "You clicked: " + $( this ).attr( "id" ));
});

// we should avoid using the DOM element to create a
// jQuery object (with the overhead that comes with it)
// and just use the DOM element itself like this:

$( "div" ).on( "click", function () {
  console.log( "You clicked:"  + this.id );
});
```

James had wanted to use jQuery's `jQuery.text` in the following context; however, he disagreed with the notion that a new jQuery object had to be created on each iteration:

```
$( "a" ).map( function () {
  return $( this ).text();
});
```

Now with respect to redundant wrapping, where possible with jQuery's utility methods, it's better to use `jQuery.methodName` (e.g., `jQuery.text`) as opposed to `jQuery.fn.methodName` (e.g., `jQuery.fn.text`), where `methodName` represents a utility such as `each()` or `text`. This avoids the need to call a further level of abstraction or construct a new jQuery object each time our function is called, as `jQuery.methodName` is what the library itself uses at a lower-level to power `jQuery.fn.methodName`.

Because though not all of jQuery's methods have corresponding single-node functions, Padolsey devised the idea of a `jQuery.single` utility.

The idea here is that a single jQuery object is created and used for each call to `jQuery.single` (effectively meaning only one jQuery object is ever created). The implementation for this can be found below and, as we're consolidating data for multiple possible objects into a more central singular structure, it is technically also a Flyweight.

```
jQuery.single = (function( o ){

   var collection = jQuery([1]);
   return function( element ) {

       // Give collection the element:
       collection[0] = element;

        // Return the collection:
       return collection;

   };
});
```

An example of this in action with chaining is:

```
$( "div" ).on( "click", function () {

   var html = jQuery.single( this ).next().html();
   console.log( html );

});
```

Although we may believe that simply caching our jQuery code may offer equivalent performance gains, Padolsey claims that `$.single` is still worth using and can perform better. That's not to say don't apply any caching at all, just be mindful that this approach can assist. For further details about `$.single`, I recommend reading Padolsey's full post.

CHAPTER 10

JavaScript MV* Patterns

In this section, we're going to review three very important architectural patterns: MVC (Model-View-Controller), MVP (Model-View-Presenter), and MVVM (Model-View-ViewModel). In the past, these patterns have been heavily used for structuring desktop and server-side applications, but it's only been in recent years that they have been applied to JavaScript.

As the majority of JavaScript developers currently using these patterns opt to use libraries such as Backbone.js for implementing an MVC/MV*-like structure, we will compare how modern solutions differ in their interpretation of MVC compared to classical takes on these patterns.

Let us first cover the basics.

MVC

MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user input. The pattern was originally designed by Trygve Reenskaug during his time working on Smalltalk-80 (1979) where it was initially called Model-View-Controller-Editor. MVC went on to be described in depth in 1995's *Design Patterns: Elements of Reusable Object-Oriented Software* a.k.a the "GoF" book, which played a role in popularizing its use.

Smalltalk-80 MVC

It's important to understand what the original MVC pattern was aiming to solve, as it has mutated quite heavily since the days of its origin. Back in the 70s, graphical user interfaces were few and far between, and a concept known as Separated Presentation began to be used as a means to make a clear division between domain objects that modelled concepts in the real world (e.g., a photo, a person) and the presentation objects that were rendered to the users screen.

The Smalltalk-80 implementation of MVC took this concept further and had an objective of separating out the application logic from the user interface. The idea was that decoupling these parts of the application would also allow the reuse of models for other interfaces in the application. There are some interesting points worth noting about Smalltalk-80's MVC architecture:

- A Model represented domain-specific data and was ignorant of the user interface (Views and Controllers). When a model changed, it would inform its observers.
- A View represented the current state of a Model. The Observer pattern was used for letting the View know whenever the Model was updated or modified.
- Presentation was taken care of by the View, but there wasn't just a single View and Controller—a View-Controller pair was required for each section or element displayed on the screen.
- The Controller's role in this pair was handling user interaction (such as key presses and actions such as clicks), making decisions for the View.

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as the Publish/Subscribe variation) was included as a part of MVC's architecture many decades ago. In Smalltalk-80's MVC, the View observes the Model. As mentioned in the bullet point above, anytime the Model changes, the Views react. A simple example of this is an application backed by stock market data. In order for the application to be useful, any change to the data in our Models should result in the View being refreshed instantly.

Martin Fowler has done an excellent job of writing about the origins of MVC over the years, and if you are interested in some further historical information about Smalltalk-80's MVC, I recommend reading his work.

MVC for JavaScript Developers

We've reviewed the 1970s, but let us now return to the here and now. In modern times, the MVC pattern has been applied to a diverse range of programming languages, including of most relevance to us: JavaScript. JavaScript now has a number of frameworks boasting support for MVC (or variations on it, which we refer to as the MV* family), allowing developers to easily add structure to their applications without great effort.

These frameworks include the likes of Backbone, Ember.js, and AngularJS. Given the importance of avoiding "spaghetti" code, a term that describes code that is very difficult to read or maintain due to its lack of structure, it's imperative that the modern JavaScript developer understand what this pattern provides. This allows us to effectively appreciate what these frameworks enable us to do differently (Figure 10-1).

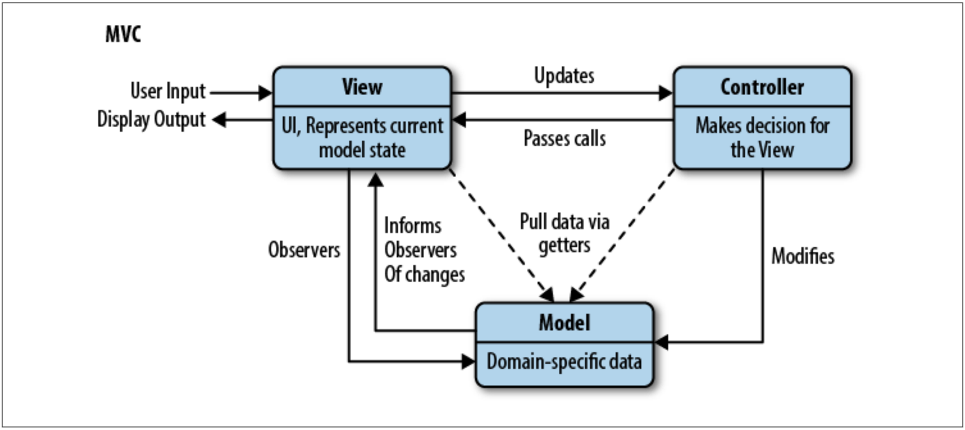


Figure 10-1. MVC pattern

We know that MVC is composed of three core components, described in the following sections.

Models

Models manage the data for an application. They are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes (e.g., when it is updated), it will typically notify its observers (e.g., views, a concept we will cover shortly) that a change has occurred so that they may react accordingly.

To understand models further, let us imagine we have a JavaScript photo gallery application. In a photo gallery, the concept of a photo would merit its own model, as it represents a unique kind of domain-specific data. Such a model may contain related attributes such as a caption, image source, and additional metadata. A specific photo would be stored in an instance of a model, and a model may also be reusable. Below, we can see an example of a very simplistic model implemented using Backbone.

```
var Photo = Backbone.Model.extend({

    // Default attributes for the photo
    defaults: {
      src: "placeholder.jpg",
      caption: "A default image",
      viewed: false
    },

    // Ensure that each photo created has an `src`.
    initialize: function() {
       this.set( { "src": this.defaults.src} );
    }

});
```

The built-in capabilities of models vary across frameworks, however it is quite common for them to support validation of attributes, where attributes represent the properties of the model, such as a model identifier. When using models in real-world applications we generally also desire model persistence. Persistence allows us to edit and update models with the knowledge that its most recent state will be saved in either memory, in a user's `localStorage` data store, or synchronized with a database.

In addition, a model may have multiple views observing it. If, say, our photo model contained metadata, such as its location (longitude and latitude), friends who were present in the photo (a list of identifiers), and a list of tags, a developer may decide to provide a single view to display each of these three facets.

It is not uncommon for modern MVC/MV* frameworks to provide a means to group models together (e.g., in Backbone, these groups are referred to as "collections"). Managing models in groups allows us to write application logic based on notifications from the group should any model it contains be changed. This avoids the need to manually observe individual model instances.

A sample grouping of models into a simplified Backbone collection can be seen here.

```
var PhotoGallery = Backbone.Collection.extend({

    // Reference to this collection's model.
    model: Photo,

    // Filter down the list of all photos
    // that have been viewed
    viewed: function() {
        return this.filter(function( photo ){
           return photo.get( "viewed" );
        });
    },

    // Filter down the list to only photos that
    // have not yet been viewed
    unviewed: function() {
      return this.without.apply( this, this.viewed() );
    }
});
```

Older texts on MVC may also refer to a notion of models managing application *state*. In JavaScript applications, *state* has a different connotation, typically referring to the current "state"—i.e., view or subview (with specific data) on a users screen at a fixed point. State is regularly discussed when looking at single-page applications, where the concept of state needs to be simulated.

So to summarize, models are primarily concerned with business data.

Views

Views are a visual representation of models that present a filtered view of their current state. While Smalltalk views are about painting and maintaining a bitmap, JavaScript views are about building and maintaining a DOM element.

A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as "dumb," given that their knowledge of models and controllers in an application is limited.

Users are able to interact with views, and this includes the ability to read and edit (i.e., get or set the attribute values in) models. As the view is the presentation layer, we generally present the ability to edit and update in a user-friendly fashion. For example, in the former photo gallery application we discussed earlier, model editing could be facilitated through an "edit" view where a user who has selected a specific photo could edit its metadata.

The actual task of updating the model falls to controllers (which we will be covering shortly).

Let's explore views a little further using a vanilla JavaScript sample implementation. Below we can see a function that creates a single photo view, consuming both a model instance and a controller instance.

We define a `render()` utility within our view, which is responsible for rendering the contents of the `photoModel` using a JavaScript templating engine (Underscore templating) and updating the contents of our view, referenced by `photoEl`.

The `photoModel` then adds our `render()` callback as one of its subscribers so that through the Observer pattern we can trigger the view to update when the model changes.

One may wonder where user interaction comes into play here. When users click on any elements within the view, it's not the view's responsibility to know what to do next. It relies on a controller to make this decision for it. In our sample implementation, this is achieved by adding an event listener to `photoEl`, which will delegate handling the click behavior back to the controller, passing the model information along with it in case it's needed.

The benefit of this architecture is that each component plays its own separate role in making the application function as needed.

```
var buildPhotoView = function ( photoModel, photoController ) {

  var base = document.createElement( "div" ),
      photoEl = document.createElement( "div" );

  base.appendChild(photoEl);

  var render = function () {
         // We use a templating library such as Underscore
```

```
            // templating which generates the HTML for our
            // photo entry
            photoEl.innerHTML = _.template( "#photoTemplate" , {
                src: photoModel.getSrc()
            });
        };

    photoModel.addSubscriber( render );

    photoEl.addEventListener( "click", function () {
      photoController.handleEvent( "click", photoModel );
    });

    var show = function () {
      photoEl.style.display = "";
    };

    var hide = function () {
      photoEl.style.display = "none";
    };

    return {
      showView: show,
      hideView: hide
    };

  };
```

Templating

In the context of JavaScript frameworks that support MVC/MV*, it is worth briefly discussing JavaScript templating and its relationship to views, as we briefly touched upon it in the last section.

It has long been considered (and proven) a performance bad practice to manually create large blocks of HTML markup in memory through string concatenation. Developers doing so have fallen prey to unperformant iterating through their data, wrapping it in nested `divs`, and using outdated techniques such as `document.write` to inject the "template" into the DOM. As this typically means keeping scripted markup inline with our standard markup, it can quickly become both difficult to read and, more importantly, maintain such disasters, especially when building nontrivially sized applications.

JavaScript templating solutions (such as Handlebars.js and Mustache) are often used to define templates for views as markup (either stored externally or within script tags with a custom type—e.g., `text/template`) containing template variables. Variables may be delimited using a variable syntax (e.g., `{{name}}`), and frameworks are typically smart enough to accept data in a JSON form (which model instances can be converted to), such that we only need be concerned with maintaining clean models and clean templates. Most of the grunt work to do with population is taken care of by the framework itself. This has a large number of benefits, particularly when opting to store templates externally, as this can give way to templates being dynamically loaded on an as-needed basis when it comes to building larger applications.

Here, we can see two examples of HTML templates (Examples 10-1 and 10-2). One implemented using the popular Handlebars.js framework and another using Underscore's templates.

Example 10-1. Handlebars.js code

```
<li class="photo">
  <h2>{{caption}}</h2>
  <img class="source" src="{{src}}"/>
  <div class="meta-data">
    {{metadata}}
  </div>
</li>
```

Example 10-2. Underscore.js Microtemplates

```
<li class="photo">
  <h2><%= caption %></h2>
  <img class="source" src="<%= src %>"/>
  <div class="meta-data">
    <%= metadata %>
  </div>
</li>
```

Note that templates are not themselves views. Developers coming from a Struts Model 2 architecture may feel like a template *is* a view, but it isn't. A view is an object that observes a model and keeps the visual representation up to date. A template *might* be a declarative way to specify part or even all of a view object so that it can be generated from the template specification.

It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In single-page JavaScript applications, however, once data is fetched from a server via Ajax, it can simply be dynamically rendered in a new view within the same page without any such refresh being necessary. The role of navigation thus falls to a router, which assists in managing application state (e.g., allowing users to bookmark a particular view they have navigated to). As routers are however neither a part of MVC nor present in every MVC-like framework, I will not be going into them in greater detail in this section.

To summarize, views are a visual representation of our application data.

Controllers

Controllers are intermediaries between models and views, which are classically responsible for updating the model when the user manipulates the view.

In our photo gallery application, a controller would be responsible for handling changes the user made to the edit view for a particular photo, updating a specific photo model when a user has finished editing.

Remember that the controllers fulfill one role in MVC: the facilitation of the Strategy pattern for the view. In the Strategy pattern regard, the view delegates to the controller at the view's discretion; that's how the strategy pattern works. The view could delegate handling user events to the controller when the view sees fit. The view *could* delegate handling model change events to the controller if the view sees fit, but this is not the traditional role of the controller.

In terms of where most JavaScript MVC frameworks detract from what is conventionally considered "MVC" however, it is with controllers. The reasons for this vary, but in my honest opinion, it is that framework authors initially look at the server-side interpretation of MVC, realize that it doesn't translate 1:1 on the client side, and reinterpret the C in MVC to mean something they feel makes more sense. The issue with this, however, is that it is subjective and increases the complexity in both understanding the classical MVC pattern and the role of controllers in modern frameworks.

As an example, let's briefly review the architecture of the popular architectural framework Backbone.js. Backbone contains models and views (somewhat similar to what we reviewed earlier); however, it doesn't actually have true controllers. Its views and routers act a little similar to a controller, but neither are actually controllers on their own.

In this respect, contrary to what might be mentioned in the official documentation or in blog posts, Backbone is neither a truly MVC/MVP nor MVVM framework. It's in fact better to consider it a member of the MV* family that approaches architecture in its own way. There is of course nothing wrong with this, but it is important to distinguish between classical MVC and MV*, should we begin relying on advice from classical literature on the former to help with the latter.

Controllers in Another Library (Spine.js) Versus Backbone.js

Spine.js

We now know that controllers are traditionally responsible for updating the model when the user updates the view. It's interesting to note that the most popular JavaScript MVC/MV* framework at the time of writing (Backbone) does not have its own explicit concept of controllers.

It can thus be useful for us to review the controller from another MVC framework to appreciate the difference in implementations and further demonstrate how nontraditionally frameworks approach the role of the controller. For this, let's take a look at a sample controller from Spine.js.

In this example, we're going to have a controller called `PhotosController`, which will be in charge of individual photos in the application. It will ensure that when the view updates (e.g., a user edited the photo metadata), the corresponding model does, too.

We won't be delving heavily into Spine.js at all, but will just take a 10-foot view of what its controllers can do:

```
// Controllers in Spine are created by inheriting from Spine.Controller

var PhotosController = Spine.Controller.sub({

  init: function () {
    this.item.bind( "update" , this.proxy( this.render ));
    this.item.bind( "destroy", this.proxy( this.remove ));
  },

  render: function () {
    // Handle templating
    this.replace( $( "#photoTemplate" ).tmpl( this.item ) );
    return this;
  },

  remove: function () {
    this.el.remove();
    this.release();
  }
});
```

In Spine, controllers are considered the glue for an application, adding and responding to DOM events, rendering templates, and ensuring that views and models are kept in sync (which makes sense in the context of what we know to be a controller).

What we're doing in the above example is setting up listeners in the `update` and `destroy` events using `render()` and `remove()`. When a photo entry gets updated, we re-render the view to reflect the changes to the metadata. Similarly, if the photo gets deleted from the gallery, we remove it from the view. In the `render()` function, we're using Underscore microtemplating (via `_.template()`) to render a JavaScript template with the ID `#photoTemplate`. This simply returns a compiled HTML string used to populate the contents of `photoEl`.

What this provides us with is a very lightweight, simple way to manage changes between the model and the view.

Backbone.js

Later in this section, we're going to revisit the differences between Backbone and traditional MVC, but for now, let's focus on controllers.

In Backbone, one shares the responsibility of a controller with both `Backbone.View` and `Backbone.Router`. Some time ago, Backbone did come with its own `Backbone.Controller`, but as the naming for this component didn't make sense for the context in which it was being used, it was later renamed to `Router`.

Routers handle a little more of the controller responsibility, as it's possible to bind the events there for models and have our view respond to DOM events and rendering. As Tim Branyen (another Bocoup-based Backbone contributor) has also previously pointed out, it's possible to get away with not needing `Backbone.Router` at all for this, so a way to think about it using the Router `paradigm` is probably:

```
var PhotoRouter = Backbone.Router.extend({
  routes: { "photos/:id": "route" },

  route: function( id ) {
    var item = photoCollection.get( id );
    var view = new PhotoView( { model: item } );

    $('.content').html( view.render().el );
  }
});
```

To summarize, the takeaway from this section is that controllers manage the logic and coordination between models and views in an application.

What Does MVC Give Us?

This separation of concerns in MVC facilitates simpler modularization of an application's functionality and enables:

- Easier overall maintenance. When updates need to be made to the application it is very clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views.
- Decoupling models and views means that it is significantly more straight-forward to write unit tests for business logic.
- Duplication of low-level model and controller code (i.e., what we may have been using instead) is eliminated across the application.
- Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user interfaces to work simultaneously.

Smalltalk-80 MVC in JavaScript

Although the majority of modern-day JavaScript frameworks attempt to evolve the MVC paradigm to better fit the differing needs of web application development, there is one framework that attempts to adhere to the pure form of the pattern found in Smalltalk-80. Maria.js (*https://github.com/petermichaux/maria*) by Peter Michaux offers an implementation that is faithful to MVC's origins: Models are models, Views are views and Controllers are nothing but controllers. While some developers might feel an MV* framework should address more concerns, this is a useful reference to be aware of in case you would like a JavaScript implementation of the original MVC.

Delving Deeper

At this point in the book, we should have a basic understanding of what the MVC pattern provides, but there's still some fascinating information about it worth noting.

The GoF (Gang of Four) The GoF do not refer to MVC as a design pattern, but rather consider it *a set of classes to build a user interface*. In their view, it's actually a variation of three classical design patterns: the Observer, Strategy, and Composite patterns. Depending on how MVC has been implemented in a framework, it may also use the Factory and Template patterns. The GoF book mentions these patterns as useful extras when working with MVC.

As we have discussed, models represent application data, while views represent what the user is presented with on screen. As such, MVC relies on the Observer pattern for some of its core communication (something that, surprisingly, isn't covered in many articles about the MVC pattern). When a model is changed, it notifies its observers (Views) that something has been updated—this is perhaps the most important relationship in MVC. The observer nature of this relationship is also what facilitates multiple views being attached to the same model.

For developers interested in knowing more about the decoupled nature of MVC (once again, depending on the implementation), one of the goals of the pattern is to help define one-to-many relationships between a topic and its observers. When a topic changes, its observers are updated. Views and controllers have a slightly different relationship. Controllers facilitate views to respond to different user input and are an example of the Strategy pattern.

Summary

Having reviewed the classical MVC pattern, we should now understand how it allows us to cleanly separate concerns in an application. We should also now appreciate how JavaScript MVC frameworks may differ in their interpretation of the MVC pattern, which although quite open to variation, still share some of the fundamental concepts the original pattern has to offer.

When reviewing a new JavaScript MVC/MV* framework, remember: it can be useful to step back and review how it's opted to approach architecture (specifically, how it supports implementing models, views, controllers or other alternatives), as this can better help us grok how the framework expects to be used.

MVP

Model View Presenter (MVP) is a derivative of the MVC design pattern that focuses on improving presentation logic. It originated at a company named Taligent in the early 1990s while they were working on a model for a C++ CommonPoint environment.

While both MVC and MVP target the separation of concerns across multiple components, there are some fundamental differences between them.

For the purposes of this summary, we will focus on the version of MVP most suitable for web-based architectures.

Models, Views, and Presenters

The P in MVP stands for presenter. It's a component that contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things, such as being able to mock views in unit tests (Figure 10-2).

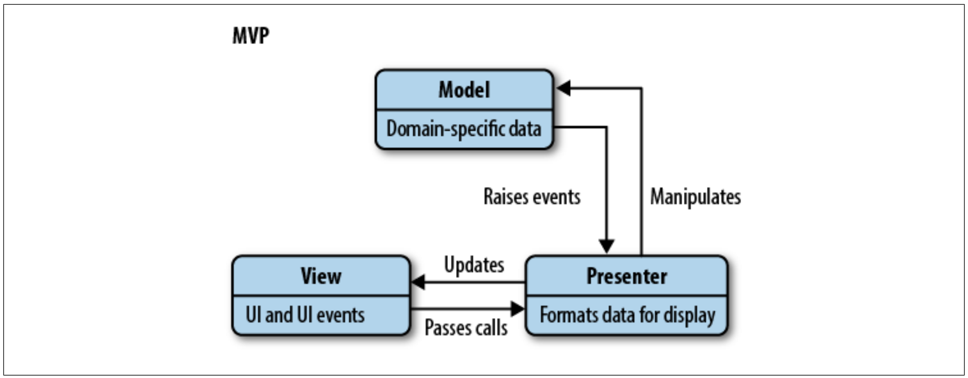


Figure 10-2. MVP pattern

The most common implementation of MVP is one that uses a Passive View (a view which is, for all intents and purposes, "dumb"), containing little to no logic. If MVC and MVP are different, it is because the C and P do different things. In MVP, the P observes models and updates views when models change. The P effectively binds models to views, a responsibility that was previously held by controllers in MVC.

Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it, and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events, but it's the presenters role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters that presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of our application and provides a more clean separation between the view and the model. This isn't however without its costs,as the lack of data binding support in the pattern can often mean having to take care of this task separately.

Although a common implementation of a Passive View is for the view to implement an interface, there are variations on it, including the use of events that can decouple the View from the Presenter a little more. As we don't have the interface construct in JavaScript, we're using more a protocol than an explicit interface here. It's technically still an API, and it's probably fair for us to refer to it as an interface from that perspective.

There is also a Supervising Controller variation of MVP, which is closer to the MVC and MVVM patterns, as it provides data-binding from the Model directly from the View. Key-value observing (KVO) plug-ins (such as Derick Bailey's `Backbone.ModelBinding` plug-in) tend to bring Backbone out of the Passive View and more into the Supervising Controller or MVVM variations.

MVP or MVC?

MVP is generally used most often in enterprise-level applications where it's necessary to reuse as much presentation logic as possible. Applications with very complex views and a great deal of user interaction may find that MVC doesn't quite fit the bill here as solving this problem may mean heavily relying on multiple controllers. In MVP, all of this complex logic can be encapsulated in a presenter, which can simplify maintenance greatly.

As MVP views are defined through an interface, and the interface is technically the only point of contact between the system and the view (other than a presenter), this pattern also allows developers to write presentation logic without needing to wait for designers to produce layouts and graphics for the application.

Depending on the implementation, MVP may be easier to automatically unit test than MVC. The reason often cited for this is that the presenter can be used as a complete mock of the user interface and so it can be unit tested independent of other components. In my experience this really depends on the languages in which we are implementing MVP (there's quite a difference between opting for MVP for a JavaScript project over one for, say, ASP.NET).

At the end of the day, the underlying concerns we may have with MVC will likely hold true for MVP, given that the differences between them are mainly semantic. As long as we are cleanly separating concerns into models, views, and controllers (or presenters), we should be achieving most of the same benefits regardless of the variation we opt for.

MVC, MVP, and Backbone.js

There are very few, if any architectural JavaScript frameworks that claim to implement the MVC or MVP patterns in their classical form, as many JavaScript developers don't view MVC and MVP as being mutually exclusive (we are actually more likely to see MVP strictly implemented when looking at web frameworks such as ASP.NET or GWT). This is because it's possible to have additional presenter/view logic in our application and still consider it a flavor of MVC.

Backbone contributor of Boston-based Bocoup Irene Ros subscribes to this way of thinking, as when she separates views out into their own distinct components, she needs something to actually assemble them for her. This could either be a controller route (such as a `Backbone.Router`, covered later in the book), or a callback in response to data being fetched.

That said, some developers do feel that Backbone.js better fits the description of MVP than it does MVC. Their view is that:

- The presenter in MVP better describes the `Backbone.View` (the layer between View templates and the data bound to it) than a controller does.
- The model fits `Backbone.Model` (it isn't greatly different to the models in MVC at all).
- The views best represent templates (e.g., Handlebars/Mustache markup templates).

A response to this could be that the view can also just be a View (as per MVC), because Backbone is flexible enough to let it be used for multiple purposes. The V in MVC and the P in MVP can both be accomplished by `Backbone.View` because they're able to achieve two purposes: both rendering atomic components and assembling those components rendered by other views.

We've also seen that in Backbone the responsibility of a controller is shared with both `Backbone.View` and `Backbone.Router` and in the following example, we can actually see that aspects of that are certainly true.

Our Backbone `PhotoView` uses the Observer pattern to "subscribe" to changes to a View's model in the line `this.model.bind("change",...)`. It also handles templating in the `render()` method, but unlike some other implementations, user interaction is also handled in the View (see `events`).

```
var PhotoView = Backbone.View.extend({

    //... is a list tag.
    tagName:  "li",

    // Pass the contents of the photo template through a templating
    // function, cache it for a single photo
    template: _.template( $("#photo-template").html() ),

    // The DOM events specific to an item.
    events: {
      "click img" : "toggleViewed"
    },

    // The PhotoView listens for changes to
    // its model, re-rendering. Since tHere's
    // a one-to-one correspondence between a
    // **Photo** and a **PhotoView** in this
    // app, we set a direct reference on the model for convenience.

    initialize: function() {
      this.model.on( "change", this.render, this );
      this.model.on( "destroy", this.remove, this );
    },

    // Re-render the photo entry
    render: function() {
      $( this.el ).html( this.template(this.model.toJSON() ));
      return this;
    },

    // Toggle the `"viewed"` state of the model.
    toggleViewed: function() {
      this.model.viewed();
    }

});
```

Another (quite different) opinion is that Backbone more closely resembles Smalltalk-80 MVC, which we went through earlier.

As regular Backbone blogger Derick Bailey has previously put it, it's ultimately best not to force Backbone to fit any specific design patterns. Design patterns should be considered flexible guides to how applications may be structured, and in this respect, Backbone fits neither MVC nor MVP. Instead, it borrows some of the best concepts from multiple architectural patterns and creates a flexible framework that just works well.

It *is* however worth understanding where and why these concepts originated, so I hope that my explanations of MVC and MVP have been of help. Call it *the Backbone way*, MV*, or whatever helps reference its flavor of application architecture. Most structural JavaScript frameworks will adopt their own take on classical patterns, either intentionally or by accident, but the important thing is that they help us develop applications that are organized, clean, and can be easily maintained.

MVVM

MVVM (Model View ViewModel) is an architectural pattern based on MVC and MVP, which attempts to more clearly separate the development of user interfaces (UI) from that of the business logic and behavior in an application. To this end, many implementations of this pattern make use of declarative data bindings to allow a separation of work on Views from other layers.

This facilitates UI and development work occurring almost simultaneously within the same code base. UI developers write bindings to the ViewModel within their document markup (HTML), where the Model and ViewModel are maintained by developers working on the logic for the application (Figure 10-3).

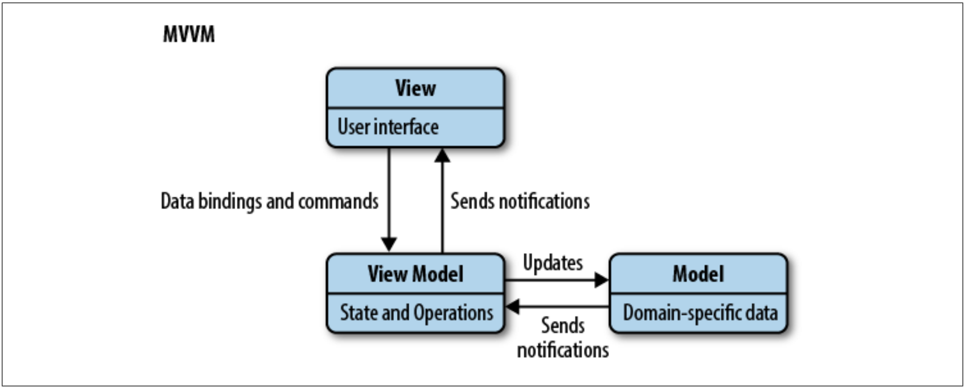


Figure 10-3. MVVM pattern

History

MVVM (by name) was originally defined by Microsoft for use with Windows Presentation Foundation (WPF) and Silverlight, having been officially announced in 2005 by John Grossman in a blog post about Avalon (the codename for WPF). It also found some popularity in the Adobe Flex community as an alternative to simply using MVC.

Prior to Microsoft adopting the MVVM name, there was however a movement in the community to go from MVP to MVPM: Model View PresentationModel. Martin Fowler wrote an article on PresentationModels back in 2004 for those interested in reading more about it. The idea of a PresentationModel had been around much longer than this article; however, it was considered the big break in the idea and greatly helped popularize it.

There was quite a lot of uproar in the "alt.net" circles after Microsoft announced MVVM as an alternative to MVPM. Many claimed the company's dominance in the GUI world was giving them the opportunity to take over the community as a whole, renaming existing concepts as they pleased for marketing purposes. A progressive crowd recognized that while MVVM and MVPM were effectively the same idea, they came in slightly different packages.

In recent years, MVVM has been implemented in JavaScript in the form of structural frameworks such as KnockoutJS, Kendo MVVM, and Knockback.js, with an overall positive response from the community.

Let's now review the three components that compose MVVM.

Model

As with other members of the MV* family, the Model in MVVM represents domain-specific data or information that our application will be working with. A typical example of domain-specific data might be a user account (e.g., name, avatar, email) or a music track (e.g., title, year, album).

Models hold information, but typically don't handle behavior. They don't format information or influence how data appears in the browser, as this isn't their responsibility. Instead, formatting of data is handled by the View, whilst behavior is considered business logic that should be encapsulated in another layer that interacts with the Model: the ViewModel.

The only exception to this rule tends to be validation, and it's considered acceptable for Models to validate data being used to define or update existing models (e.g., does an email address being input meet the requirements of a particular regular expression?).

In KnockoutJS, Models fall under the above definition, but often make Ajax calls to a server-side service to both read and write Model data.

If we were constructing a simple Todo application, a KnockoutJS model representing a single Todo item could look as follows:

```
var Todo = function ( content, done ) {
    this.content = ko.observable(content);
    this.done = ko.observable(done);
    this.editing = ko.observable(false);
};
```

You might notice in the above snippet that we are calling the method `observable()` on the KnockoutJS namespace `ko`. In KnockoutJS, observables are special JavaScript objects that can notify subscribers about changes and automatically detect dependencies. This allows us to synchronize Models and ViewModels when the value of a Model attribute is modified.

View

As with MVC, the View is the only part of the application that users actually interact with. They are an interactive UI that represents the state of a ViewModel. In this sense, the view is considered active rather than passive, but this is also true for views in MVC and MVP. In MVC, MVP, and MVVM, a view can also be passive, but what does this mean?

A passive View only outputs a display and does not accept any user input. Such a view may also have no real knowledge of the models in our application and could be manipulated by a presenter. MVVM's active View contains the data bindings, events, and behaviors, which requires an understanding of the ViewModel. Although these behaviors can be mapped to properties, the View is still responsible for handling events from the ViewModel.

It's important to remember the View isn't responsible here for handling state; it keeps this in sync with the ViewModel.

A KnockoutJS View is simply a HTML document with declarative bindings to link it to the ViewModel. KnockoutJS Views display information from the ViewModel, pass commands to it (e.g., a user clicking on an element), and update as the state of the ViewModel changes. Templates generating markup using data from the ViewModel can however also be used for this purpose.

To give a brief initial example, we can look to the JavaScript MVVM framework KnockoutJS for how it allows the definition of a ViewModel and its related bindings in markup.

Here is the code for the ViewModel:

```
var aViewModel = {
    contactName: ko.observable( "John" );
};
```

Here is the code for the View:

```
<input id="source" data-bind="value: contactName, valueUpdate: "keyup" /></p>

<div data-bind="visible: contactName().length > 10">
    You have a really long name!
</div>
```

Our input text box (source) obtains its initial value from `contactName`, automatically updating this value whenever contactName changes. As the data binding is two-way, typing into the text box will update `contactName` accordingly so the values are always in sync.

Although implementation specific to KnockoutJS, the `<div>` containing the "You have a really long name!" text also contains simple validation (once again in the form of data bindings). If the input exceeds 10 characters, it will display; otherwise, it will remain hidden.

We can return to our Todo application for a more advanced example. A trimmed down KnockoutJS View, including all the necessary data bindings, may look as follows:

```
<div id="todoapp">
    <header>
        <h1>Todos</h1>
        <input id="new-todo" type="text" data-bind="value: current,
        valueUpdate: "afterkeydown", enterKey: add"
               placeholder="What needs to be done?"/>
    </header>
    <section id="main" data-bind="block: todos().length">

        <input id="toggle-all" type="checkbox" data-bind="checked: allCompleted">
        <label for="toggle-all">Mark all as complete</label>

        <ul id="todo-list" data-bind="foreach: todos">

           <!-- item -->
            <li data-bind="css: { done: done, editing: editing }">
                <div class="view" data-bind="event: { dblclick: $root.editItem }">
```

```
                    <input class="toggle" type="checkbox" data-bind="checked: done">
                    <label data-bind="text: content"></label>
                    <a class="destroy" href="#" data-bind="click: $root.remove"></a>
                </div>
                <input class="edit' type="text"
                       data-bind="value: content, valueUpdate: "afterkeydown",
                       enterKey: $root.stopEditing, selectAndFocus: editing,
                       event: { blur: $root.stopEditing }"/>
            </li>

        </ul>

    </section>
</div>
```

Note that the basic layout of the markup is relatively straightforward, containing an input text box (`new-todo`) for adding new items, togglers for marking items as complete, and a list (`todo-list`) with a template for a Todo item in the form of an `li`.

The data bindings in the above markup can be broken down as follows:

- The input text box `new-todo` has a data binding for the `current` property, which is where the value of the current item being added is stored. Our ViewModel (shown shortly) observes the `current` property and also has a binding against the `add` event. When the Enter key is pressed, the `add` event is triggered, and our ViewModel can then trim the value of `current` and add it to the Todo list as needed.
- The input checkbox `toggle-all` can mark all of the current items as completed if clicked. If checked, it triggers the `allCompleted` event, which can be seen in our ViewModel.
- The item `li` has the class `done`. When a task is marked as done, the CSS class `editing` is marked accordingly. If double-clicking on the item, the `$root.editI tem` callback will be executed.
- The checkbox with the class `toggle` shows the state of the `done` property.
- A label contains the text value of the Todo item (`content`).
- There is also a remove button that will call the `$root.remove` callback when clicked.
- An input text box used for editing mode also holds the value of the Todo item `content`. The `enterKey` event will set the `editing` property to true or false.

ViewModel

The ViewModel can be considered a specialized Controller that acts as a data converter. It changes Model information into View information, passing commands from the View to the Model.

For example, let us imagine that we have a model containing a `date` attribute in UNIX format (e.g., 1333832407). Rather than our models being aware of a user's view of the date (e.g., 04/07/2012 @ 5:00pm), where it would be necessary to convert the address to its display format, our model simply holds the raw format of the data. Our View contains the formatted date, and our ViewModel acts as a middleman between the two.

In this sense, the ViewModel might be looked upon as more of a Model than a View, but it does handle most of the View's display logic. The ViewModel may also expose methods for helping to maintain the View's state, update the model based on the actions on a View, and trigger events on the View.

In summary, the ViewModel sits behind our UI layer. It exposes data needed by a View (from a Model) and can be viewed as the source our Views go to for both data and actions.

KnockoutJS interprets the ViewModel as the representation of data and operations that can be performed on a UI. This isn't the UI itself nor the data model that persists, but rather a layer that can also hold the yet to be saved data a user is working with. Knockout's ViewModels are implemented JavaScript objects with no knowledge of HTML markup. This abstract approach to their implementation allows them to stay simple, meaning more complex behavior can be more easily managed on top as needed.

A partial KnockoutJS ViewModel for our Todo application could thus look as follows:

```
// our main ViewModel
    var ViewModel = function ( todos ) {
        var self = this;

    // map array of passed in todos to an observableArray of Todo objects
    self.todos = ko.observableArray(
    ko.utils.arrayMap( todos, function ( todo ) {
        return new Todo( todo.content, todo.done );
    }));

    // store the new todo value being entered
    self.current = ko.observable();

    // add a new todo, when enter key is pressed
    self.add = function ( data, event ) {
        var newTodo, current = self.current().trim();
        if ( current ) {
            newTodo = new Todo( current );
            self.todos.push( newTodo );
            self.current("");
        }
    };

    // remove a single todo
    self.remove = function ( todo ) {
        self.todos.remove( todo );
    };

    // remove all completed todos
    self.removeCompleted = function () {
        self.todos.remove(function (todo) {
            return todo.done();
        });
    };

    // writeable computed observable to handle marking all complete/incomplete
    self.allCompleted = ko.computed({

        // always return true/false based on the done flag of all todos
        read:function () {
            return !self.remainingCount();
        },

        // set all todos to the written value (true/false)
        write:function ( newValue ) {
            ko.utils.arrayForEach( self.todos(), function ( todo ) {
                // set even if value is the same, as
                subscribers are not notified in that case
                todo.done( newValue );
            });
        }
    });

    // edit an item
    self.editItem = function( item ) {
        item.editing( true );
    };
..
```

Here, we are basically providing the methods needed to add, edit, or remove items as well as the logic to mark all remaining items as having been completed. Note that the only real difference from previous examples in our ViewModel are observable arrays. In KnockoutJS, if we wish to detect and respond to changes on a single object, we would use `observables`. If, however, we wish to detect and respond to changes of a collection of things, we can use an `observableArray` instead. A simpler example of how to use observable arrays may look as follows:

```
// Define an initially an empty array
var myObservableArray = ko.observableArray();

// Add a value to the array and notify our observers
myObservableArray.push( 'A new todo item' );
```

The complete Knockout.js Todo application we reviewed above can be grabbed from TodoMVC.

Recap: The View and the ViewModel

Views and ViewModels communicate using data bindings and events. As we saw in our initial ViewModel example, the ViewModel doesn't just expose Model attributes but also access to other methods and features such as validation.

Our Views handle their own user interface events, mapping them to the ViewModel as necessary. Models and attributes on the ViewModel are synchronized and updated via two-way data binding.

Triggers (data triggers) also allow us to further react to changes in the state of our Model attributes.

Recap: The ViewModel and the Model

While it may appear that the ViewModel is completely responsible for the Model in MVVM, there are some subtleties with this relationship worth noting. The ViewModel can expose a Model or Model attributes for the purposes of data binding and can also contain interfaces for fetching and manipulating properties exposed in the view.

Pros and Cons

We now hopefully have a better appreciation for what MVVM is and how it works. Let's now review the advantages and disadvantages of employing the pattern.

Advantages

- MVVM facilitates easier parallel development of a UI and the building blocks that power it.
- MVVM abstracts the View and thus reduces the quantity of business logic (or glue) required in the code behind it.
- The ViewModel can be easier to unit test than in the case of event-driven code.
- The ViewModel (being more Model than View) can be tested without concerns of UI automation and interaction.

Disadvantages

- For simpler UIs, MVVM can be overkill.
- While data bindings can be declarative and nice to work with, they can be harder to debug than imperative code where we simply set breakpoints.
- Data bindings in nontrivial applications can create a lot of bookkeeping. We also don't want to end up in a situation where bindings are heavier than the objects being bound to.
- In larger applications, it can be more difficult to design the ViewModel up front to get the necessary amount of generalization.

MVVM with Looser Data Bindings

It’s not uncommon for JavaScript developers from an MVC or MVP background to review MVVM and complain about its true separation of concerns. Namely, the quantity of inline data bindings maintained in the HTML markup of a View.

I must admit that when I first reviewed implementations of MVVM (e.g., KnockoutJS, Knockback), I was surprised that any developer would want to return to the days of old where we mixed logic (JavaScript) with our markup and found it quickly unmaintainable. The reality however is that MVVM does this for a number of good reasons (which we’ve covered), including facilitating designers to more easily bind to logic from their markup.

For the purists among us, you’ll be happy to know that we can now also greatly reduce how reliant we are on data bindings, thanks to a feature known as custom binding providers, introduced in KnockoutJS 1.3 and available in all versions since.

KnockoutJS by default has a data-binding provider, which searches for any elements with `data-bind` attributes on them such as in the below example.

```
<input id="new-todo" type="text" data-bind="value: current, valueUpdate: "afterkeydown",
enterKey: add" placeholder="What needs to be done?"/>
```

When the provider locates an element with this attribute, it parses it and turns it into a binding object using the current data context. This is the way KnockoutJS has always worked, allowing us to declaratively add bindings to elements that KnockoutJS binds to the data at that layer.

Once we start building Views that are no longer trivial, we may end up with a large number of elements and attributes whose bindings in markup can become difficult to manage. With custom binding providers, however, this is no longer a problem.

A binding provider is primarily interested in two things:

- When given a DOM node, does it contain any data bindings?
- If the node passed this first question, what does the binding object look like in the current data context?

Binding providers implement two functions:

nodeHasBindings
: This takes in a DOM node, which doesn’t necessarily have to be an element.

getbindings
: This returns an object representing the bindings as applied to the current data context.

A skeleton binding provider might thus look as follows:

```
var ourBindingProvider = {
  nodeHasBindings: function( node ) {
      // returns true/false
  },

  getBindings: function( node, bindingContext ) {
      // returns a binding object
  }
};
```

Before we get to fleshing out this provider, let's briefly discuss logic in data bind attributes.

If when using Knockout's MVVM you find yourself dissatisfied with the idea of application logic being overly tied into your View, we can change this. We could implement something a little like CSS classes to assign bindings by name to elements. Ryan Niemeyer has previously suggested using `data-class` for this to avoid confusing presentation classes with data classes, so let's get our `nodeHasBindings` function supporting this:

```
    // does an element have any bindings?
    function nodeHasBindings( node ) {
        return node.getAttribute ? node.getAttribute("data-class") : false;
    };
```

Next, we need a sensible `getBindings()` function. As we're sticking with the idea of CSS classes, why not also consider supporting space-separated classes to allow us to share binding specs between different elements?

Let's first review what our bindings will look like. We create an object to hold them where our property names need to match the keys we wish to use in our data classes.

There isn't a great deal of work required to convert a KnockoutJS application from one that uses traditional data bindings to one with unobtrusive bindings by applying custom binding providers. We simply pull our all of our `data-bind` attributes, replace them with `data-class` attributes, and place our bindings in a binding object as per below:

```
    var viewModel = new ViewModel( todos || [] ),
        bindings = {

            newTodo:  {
                value: viewModel.current,
                valueUpdate: "afterkeydown",
                enterKey: viewModel.add
            },

            taskTooltip : { visible: viewModel.showTooltip },
            checkAllContainer : {visible: viewModel.todos().length },
            checkAll: { checked: viewModel.allCompleted },

            todos: { foreach: viewModel.todos },
            todoListItem: function() { return { css: { editing: this.editing } }; },
            todoListItemWrapper: function() {
              return { css: { done: this.done } };
            },
            todoCheckBox: function() { return { checked: this.done }; },
            todoContent: function() { return { text: this.content, event:
            { dblclick: this.edit } }; },
            todoDestroy: function() { return { click: viewModel.remove }; },

            todoEdit: function() { return {
                value: this.content,
                valueUpdate: "afterkeydown",
                enterKey: this.stopEditing,
                event: { blur: this.stopEditing } };
            },

            todoCount: { visible: viewModel.remainingCount },
            remainingCount: { text: viewModel.remainingCount },
            remainingCountWord: function() {
              return { text: viewModel.getLabel(viewModel.remainingCount) };
            },
            todoClear: {visible: viewModel.completedCount},
            todoClearAll: {click: viewModel.removeCompleted},
            completedCount: { text: viewModel.completedCount },
            completedCountWord: function() {
              return { text: viewModel.getLabel(viewModel.completedCount) };
            },
            todoInstructions: { visible: viewModel.todos().length }
        };

        ....
```

There are however two lines missing from the above snippet: we still need our `getBindings` function, which will loop through each of the keys in our `data-class` attributes and build up the resulting object from each of them. If we detect that the binding object is a function, we call it with our current data using the context `this`. Our complete custom binding provider would look as follows:

```
// We can now create a bindingProvider that uses
// something different than data-bind attributes
ko.customBindingProvider = function( bindingObject ) {
    this.bindingObject = bindingObject;

    // determine if an element has any bindings
    this.nodeHasBindings = function( node ) {
        return node.getAttribute ? node.getAttribute( "data-class" ) : false;
    };
  };

// return the bindings given a node and the bindingContext
this.getBindings = function( node, bindingContext ) {

    var result = {},
        classes = node.getAttribute( "data-class" );

    if ( classes ) {
        classes = classes.split( "" );

        //evaluate each class, build a single object to return
        for ( var i = 0, j = classes.length; i < j; i++ ) {

           var bindingAccessor = this.bindingObject[classes[i]];
```

```
                    if ( bindingAccessor ) {
                        var binding = typeof bindingAccessor ===
                        "function" ? bindingAccessor.call(bindingContext.$data) : bindingAccessor;
                        ko.utils.extend(result, binding);
                    }

                }
            }

            return result;
        };
    };
```

Thus, the final few lines of our `bindings` object can be defined as follows:

```
    // set ko's current bindingProvider equal to our new binding provider
    ko.bindingProvider.instance = new ko.customBindingProvider( bindings );

    // bind a new instance of our ViewModel to the page
    ko.applyBindings( viewModel );

    })();
```

What we're doing here is effectively defining a constructor for our binding handle, which accepts an object (bindings) that we use to lookup our bindings. We could then rewrite the markup for our application View using data classes as follows:

```
    <div id="create-todo">
                    <input id="new-todo" data-class="newTodo" placeholder=
                    "What needs to be done?" />
                    <span class="ui-tooltip-top" data-class="taskTooltip" style=
                    "display: none;">
                    Press Enter to save this task</span>
                </div>
                <div id="todos">
                    <div data-class="checkAllContainer" >
                        <input id="check-all" class="check" type="checkbox" data-class=
                        "checkAll" />
                        <label for="check-all">Mark all as complete</label>
                    </div>
                    <ul id="todo-list" data-class="todos" >
                        <li data-class="todoListItem" >
                            <div class="todo" data-class="todoListItemWrapper" >
                                <div class="display">
                                    <input class="check" type="checkbox" data-class=
                                    "todoCheckBox" />
                                    <div class="todo-content" data-class="todoContent"
                                    style="cursor: pointer;"></div>
                                    <span class="todo-destroy" data-class=
                                    "todoDestroy"></span>
                                </div>
                                <div class="edit'>
                                    <input class="todo-input" data-class="todoEdit'/>
                                </div>
                            </div>
                        </li>
```

```
            </ul>
        </div>
```

Neil Kerkin has put together a complete TodoMVC demo app using the above, which you can access and play around with here.

While it may look like quite a lot of work in the explanation above, now that we have a generic `getBindings` method written, it's a lot more trivial to simply reuse it and use data classes rather than strict data bindings for writing our KnockoutJS applications instead. The net result is hopefully cleaner markup with our data bindings being shifted from the View to a bindings object instead.

MVC Versus MVP Versus MVVM

Both MVP and MVVM are derivatives of MVC. The key difference between MVC and its derivatives is the dependency each layer has on other layers, as well as how tightly bound they are to each other.

In MVC, the View sits on top of our architecture with the controller beside it. Models sit below the controller, so our Views know about our controllers and controllers know about Models. Here, our Views have direct access to Models. Exposing the complete Model to the View, however, may have security and performance costs, depending on the complexity of our application. MVVM attempts to avoid these issues.

In MVP, the role of the controller is replaced with a Presenter. Presenters sit at the same level as views, listening to events from both the View and model, and mediating the actions between them. Unlike MVVM, there isn't a mechanism for binding Views to ViewModels, so we instead rely on each View implementing an interface allowing the Presenter to interact with the View.

MVVM consequently allows us to create View-specific subsets of a Model, which can contain state and logic information, avoiding the need to expose the entire Model to a View. Unlike MVP's Presenter, a ViewModel is not required to reference a View. The View can bind to properties on the ViewModel, which in turn expose data contained in Models to the View. As we've mentioned, the abstraction of the View means there is less logic required in the code behind it.

One of the downsides to this however is that a level of interpretation is needed between the ViewModel and the View, and this can have performance costs. The complexity of this interpretation can also vary: it can be as simple as copying data or as complex as manipulating it to a form we would like the View to see. MVC doesn't have this problem, as the whole Model is readily available and such manipulation can be avoided.

Backbone.js Versus KnockoutJS

Understanding the subtle differences between MVC, MVP, and MVVM is important, but developers ultimately will ask whether they should consider using KnockoutJS over Backbone based in what we've learned. The following notes may be of help here:

- Both libraries are designed with different goals in mind, and it's often not as simple as just choosing MVC or MVVM.
- If data binding and two-way communication are your main concerns, KnockoutJS is definitely the way to go. Practically any attribute or value stored in DOM nodes can be mapped to JavaScript objects with this approach.
- Backbone excels with its ease of integration with RESTful services, while KnockoutJS Models are simply JavaScript objects and code needed for updating the Model must be written by the developer.
- KnockoutJS has a focus on automating UI bindings, which requires significantly more verbose custom code if attempting to do this with Backbone. This isn't a problem with Backbone itself per se, as it purposefully attempts to stay out of the UI. Knockback does, however, attempt to assist with this problem.
- With KnockoutJS, we can bind our own functions to ViewModel observables, which are executed any time the observable changes. This allows us the same level of flexibility as can be found in Backbone.
- Backbone has a solid routing solution built in, while KnockoutJS offers no routing options out of the box. One can, however, easily fill in this behavior if needed using Ben Alman's BBQ plug-in or a standalone routing system like Miller Medeiros's excellent Crossroads.

To conclude, I personally find KnockoutJS more suitable for smaller applications, while Backbone's feature set really shines when building anything nontrivial. That said, many developers have used both frameworks to write applications of varying complexity, and I recommend trying out both at a smaller scale before making a decision on which might work best for your project.

For further reading about MVVM or Knockout, I recommend the following articles:

- The Advantages Of MVVM
- SO: What are the problems with MVVM?
- MVVM Explained
- How does MVVM compare to MVC?
- Custom bindings in KnockoutJS
- Exploring Knockout with TodoMVC

CHAPTER 11

Modern Modular JavaScript Design Patterns

In the world of scalable JavaScript, when we say an application is *modular*, we often mean it's composed of a set of highly decoupled, distinct pieces of functionality stored in modules. Loose coupling facilitates easier maintainability of apps by removing *dependencies* where possible. When this is implemented efficiently, it's quite easy to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages, the current iteration of JavaScript—ECMA-262—doesn't provide developers with the means to import such modules of code in a clean, organized manner. It's one of the concerns with specifications that hasn't required great thought until more recent years when the need for more organized JavaScript applications became apparent.

Instead, developers at present are left to fall back on variations of the module or object literal patterns, which we covered earlier in the book. With many of these, module scripts are strung together in the DOM with namespaces being described by a single global object where it's still possible to incur naming collisions in our architecture. There's also no clean way to handle dependency management without some manual effort or third-party tools.

While native solutions to these problems will be arriving in ES Harmony—likely to be the next version of JavaScript—the good news is that writing modular JavaScript has never been easier, and we can start doing it today.

In this section, we're going to look at three formats for writing modular JavaScript: *AMD*, *CommonJS*, and proposals for the next version of JavaScript, *Harmony*.

A Note on Script Loaders

It's difficult to discuss AMD and CommonJS modules without talking about the elephant in the room: script loaders. At the time of writing this book, script loading is a means to a goal, that goal being modular JavaScript that can be used in applications today—for this, use of a compatible script loader is unfortunately necessary. To get the most out of this section, I recommend gaining a *basic understanding* of how popular script loading tools work so the explanations of module formats make sense in context.

There are a number of great loaders for handling module loading in the AMD and CommonJS formats, but my personal preferences are RequireJS and curl.js. Complete tutorials on these tools are outside the scope of this book, but I recommend reading John Hann's article about curl.js and James Burke's API documentation RequireJS for more.

From a production perspective, the use of optimization tools (like the RequireJS optimizer) to concatenate scripts is recommended for deployment when working with such modules. Interestingly, with the Almond AMD shim, RequireJS doesn't need to be rolled in the deployed site, and what one might consider a script loader can be easily shifted outside of development.

That said, James Burke would probably say that being able to dynamically load scripts after page load still has its use cases, and RequireJS can assist with this, too. With these notes in mind, let's get started.

AMD

The overall goal for the Asynchronous Module Definition (AMD) format is to provide a solution for modular JavaScript that developers can use today. It was born out of Dojo's real world experience using XHR+eval, and proponents of this format wanted to avoid any future solutions suffering from the weaknesses of those in the past.

The AMD module format itself is a proposal for defining modules in which both the module and dependencies can be asynchronously loaded. It has a number of distinct advantages, including being both asynchronous and highly flexible by nature, which removes the tight coupling one might commonly find between code and module identity. Many developers enjoy using it, and one could consider it a reliable stepping stone toward the module system proposed for ES Harmony.

AMD began as a draft specification for a module format on the CommonJS list, but as it wasn't able to reach full consensus, further development of the format moved to the amdjs group.

Today, it's embraced by projects including Dojo, MooTools, Firebug, and even jQuery . Although the term *CommonJS AMD format* has been seen in the wild on occasion, it's best to refer to it as just AMD or Async Module support, as not all participants on the CommonJS list wished to pursue it.

There was a time when the proposal was referred to as Modules Transport/C, however as the spec wasn't geared toward transporting existing CommonJS modules, but rather—for defining modules—it made more sense to opt for the AMD naming convention.

Getting Started with Modules

The first two concepts worth noting about AMD are the idea of a `define` method for facilitating module definition and a `require` method for handling dependency loading. `define` is used to define named or unnamed modules based using the following signature:

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

As we can tell by the inline comments, the `module_id` is an optional argument that is typically only required when non-AMD concatenation tools are being used (there may be some other edge cases where it's useful, too). When this argument is left out, we refer to the module as *anonymous*.

When working with anonymous modules, the idea of a module's identity is DRY, making it trivial to avoid duplication of filenames and code. Because the code is more portable, it can be easily moved to other locations (or around the filesystem) without needing to alter the code itself or change its module ID. Consider the `module_id` similar to the concept of folder paths.

Developers can run this same code on multiple environments just by using an AMD optimizer that works with a CommonJS environment such as r.js.

Back to the `define` signature, the `dependencies` argument represents an array of dependencies required by the module we are defining, and the third argument (`definition function` or `factory function`) is a function that's executed to instantiate our module. A bare bone module could be defined as follows (Example 11-1):

Example 11-1. Understanding AMD: define()

```
// A module_id (myModule) is used here for demonstration purposes only
define( "myModule",

    ["foo", "bar"],

    // module definition function
    // dependencies (foo and bar) are mapped to function parameters
    function ( foo, bar ) {
        // return a value that defines the module export
        // (i.e the functionality we want to expose for consumption)

        // create your module here
        var myModule = {
            doStuff:function () {
                console.log( "Yay! Stuff" );
            }
        };

    return myModule;
});

// An alternative version could be..
define( "myModule",

    ["math", "graph"],

    function ( math, graph ) {

        // Note that this is a slightly different pattern
        // With AMD, it's possible to define modules in a few
        // different ways due to it's flexibility with
        // certain aspects of the syntax
        return {
            plot: function( x, y ){
                return graph.drawPie( math.randomGrid( x, y ) );
            }
        };
});
```

`require`, on the other hand, is typically used to load code in a top-level JavaScript file or within a module, should we wish to dynamically fetch dependencies. Here is an example of its usage (Example 11-2):

Example 11-2. Understanding AMD: require()

```
// Consider "foo" and "bar" are two external modules
// In this example, the "exports" from the two modules
// loaded are passed as function arguments to the
// callback (foo and bar) so that they can similarly be accessed

require(["foo", "bar"], function ( foo, bar ) {
        // rest of your code here
        foo.doSomething();
});
```

The following is an example of a dynamically-loaded dependency (Example 11-3):

Example 11-3. Dynamically-loaded dependencies

```
define(function ( require ) {
    var isReady = false, foobar;

    // note the inline require within our module definition
    require(["foo", "bar"], function ( foo, bar ) {
        isReady = true;
        foobar = foo() + bar();
    });

    // we can still return a module
    return {
        isReady: isReady,
        foobar: foobar
    };
});
```

The following is an example of defining an AMD-compatible plug-in (Example 11-4):

Example 11-4. Understanding AMD: plug-ins

```
// With AMD, it's possible to load in assets of almost any kind
// including text-files and HTML. This enables us to have template
// dependencies which can be used to skin components either on
// page-load or dynamically.

define( ["./templates", "text!./template.md","css!./template.css" ],

    function( templates, template ){
        console.log( templates );
        // do something with our templates here
    }

});
```

> Although `css!` is included for loading CSS dependencies in the above example, it's important to remember that this approach has some caveats, such as it not being fully possible to establish when the CSS is fully loaded. Depending on how we approach our build process, it may also result in CSS being included as a dependency in the optimized file, so use CSS as a loaded dependency in such cases with caution. If interested in doing the above, we can also explore @VIISON's RequireJS CSS plug-in further here.

This example could simply be looked at as `requirejs(["app/myModule"], function() {})`, which indicates the loader's top level globals are being used. This is how to kick off top-level loading of modules with different AMD loaders; however, with a `define()` function, if it's passed a local require all `require([])` examples apply to both types of loader, curl.js and RequireJS (Examples 11-5 and 11-6).

Example 11-5. Loading AMD modules using RequireJS

```
require(["app/myModule"],

    function( myModule ){
        // start the main module which in-turn
        // loads other modules
        var module = new myModule();
        module.doStuff();
});
```

Example 11-6. Loading AMD modules using curl.js

```
curl(["app/myModule.js"],

    function( myModule ){
        // start the main module which in-turn
        // loads other modules
        var module = new myModule();
        module.doStuff();

});
```

Here is the code for modules with deferred dependencies:

```
// This could be compatible with jQuery's Deferred implementation,
// futures.js (slightly different syntax) or any one of a number
// of other implementations

define(["lib/Deferred"], function( Deferred ){
    var defer = new Deferred();

    require(["lib/templates/?index.html","lib/data/?stats"],
        function( template, data ){
            defer.resolve( { template: template, data:data } );
        }
    );
    return defer.promise();
});
```

AMD Modules with Dojo

Defining AMD-compatible modules using Dojo is fairly straightforward. As per above, define any module dependencies in an array as the first argument and provide a callback (factory) that will execute the module once the dependencies have been loaded. For example:

```
define(["dijit/Tooltip"], function( Tooltip ){

    //Our dijit tooltip is now available for local use
    new Tooltip(...);

});
```

Note the anonymous nature of the module, which can now be both consumed by a Dojo asynchronous loader, RequireJS or the standard dojo.require() module loader.

There are some interesting gotchas with module referencing that are useful to know here. Although the AMD-advocated way of referencing modules declares them in the dependency list with a set of matching arguments, this isn't supported by the older Dojo 1.6 build system—it really only works for AMD-compliant loaders. For example:

```
    define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){

        var cookieValue = cookie( "cookieName" );
        new Tooltip(...);

    });
```

This has many advantages over nested namespacing , as modules no longer need to directly reference complete namespaces every time; all we require is the `dojo/cookie` path in dependencies, which once aliased to an argument, can be referenced by that variable. This removes the need to repeatedly type out `dojo.` in our applications.

The final gotcha to be aware of is that if we wish to continue using the older Dojo build system or wish to migrate older modules to this newer AMD-style, the following more verbose version enables easier migration. Notice that `dojo` and `dijit` and referenced as dependencies, too:

```
    define(["dojo", "dijit', "dojo/cookie", "dijit/Tooltip"], function( dojo, dijit ){
        var cookieValue = dojo.cookie( "cookieName" );
        new dijit.Tooltip(...);
    });
```

AMD Module Design Patterns (Dojo)

As we've seen in previous sections, design patterns can be highly effective in improving how we approach structuring solutions to common development problems. John Hann has given some excellent presentations about AMD module design patterns covering the Singleton, Decorator, Mediator, and others, and I highly recommend checking out his slides.

A selection of AMD design patterns can be found below (Examples 11-7 and 11-8):

Example 11-7. Decorator pattern

```
// mylib/UpdatableObservable: a Decorator for dojo/store/Observable
define(["dojo", "dojo/store/Observable"], function ( dojo, Observable ) {
    return function UpdatableObservable ( store ) {

        var observable = dojo.isFunction( store.notify ) ? store :
                new Observable(store);

        observable.updated = function( object ) {
            dojo.when( object, function ( itemOrArray) {
                dojo.forEach( [].concat(itemOrArray), this.notify, this );
            });
        };

        return observable;
    };
});


// Decorator consumer
// a consumer for mylib/UpdatableObservable

define(["mylib/UpdatableObservable"], function ( makeUpdatable ) {
    var observable,
        updatable,
        someItem;

    // make the observable store updatable
    updatable = makeUpdatable( observable ); // `new` is optional!

    // we can then call .updated() later on if we wish to pass
    // on data that has changed
    //updatable.updated( updatedItem );
});
```

Example 11-8. Adapter pattern

```
// "mylib/Array" adapts `each` function to mimic jQuerys:
define(["dojo/_base/lang", "dojo/_base/array"], function ( lang, array ) {
    return lang.delegate( array, {
        each: function ( arr, lambda ) {
            array.forEach( arr, function ( item, i ) {
                lambda.call( item, i, item ); // like jQuery's each
            });
        }
    });
});

// Adapter consumer
// "myapp/my-module":
define(["mylib/Array"], function ( array ) {
    array.each( ["uno", "dos", "tres"], function ( i, esp ) {
        // here, `this` == item
    });
});
```

AMD Modules with jQuery

Unlike Dojo, jQuery really only comes with one file; however, given the plug-in-based nature of the library, we can demonstrate how straightforward it is to define an AMD module that uses it below.

```
define(["js/jquery.js","js/jquery.color.js","js/underscore.js"],

    function( $, colorPlugin, _ ){
        // Here we've passed in jQuery, the color plugin and Underscore
        // None of these will be accessible in the global scope, but we
        // can easily reference them below.

        // Pseudo-randomize an array of colors, selecting the first
        // item in the shuffled array
        var shuffleColor = _.first( _.shuffle( "#666","#333","#111"] ) );

        // Animate the background-color of any elements with the class
        // "item" on the page using the shuffled color
        $( ".item" ).animate( {"backgroundColor": shuffleColor } );

        // What we return can be used by other modules
        return {};
    });
```

There is, however, something missing from this example: the concept of registration.

Registering jQuery as an Async-compatible module

One of the key features that landed in jQuery 1.7 was support for registering jQuery as an asynchronous module. There are a number of compatible script loaders (including RequireJS and curl) that are capable of loading modules using an asynchronous module format, and this means fewer hacks are required to get things working.

If a developer wants to use AMD and does not want her jQuery version leaking into the global space, she should call `noConflict` in their top level module that uses jQuery. In addition, since multiple versions of jQuery can be on a page, there are special considerations that an AMD loader must account for, and so jQuery only registers with AMD loaders that have recognized these concerns, which are indicated by the loader specifying `define.amd.jQuery`. RequireJS and curl are two loaders that do so.

The named AMD provides a safety blanket of being both robust and safe for most use cases.

```
// Account for the existence of more than one global
// instances of jQuery in the document, cater for testing
// .noConflict()

var jQuery = this.jQuery || "jQuery",
$ = this.$ || "$",
originaljQuery = jQuery,
original$ = $;

define(["jquery"] , function ( $ ) {
    $( ".items" ).css( "background","green" );
    return function () {};
});
```

Why AMD is a better choice for writing modular JavaScript

We have now reviewed several code samples taking us through what AMD is capable of. It certainly appears to give us more than just a typical Module pattern, but why is it a better choice for modular application development?

- Provides a clear proposal for how to approach defining flexible modules.
- Significantly cleaner than the present global namespace and `<script>` tag solutions many of us rely on. There's a clean way to declare standalone modules and dependencies they may have.
- Module definitions are encapsulated, helping us to avoid pollution of the global namespace.

- Arguably works better than some alternative solutions (e.g., CommonJS, which we'll be looking at shortly). It doesn't have issues with cross-domain, local, or debugging and doesn't rely on server-side tools to be used. Most AMD loaders support loading modules in the browser without a build process.
- Provides a "transport" approach for including multiple modules in a single file. Other approaches like CommonJS have yet to agree on a transport format.
- It's possible to lazy load scripts if this is needed.

Many of the above could be said about YUI's module loading strategy as well.

Related reading

The RequireJS Guide To AMD

What's the fastest way to load AMD modules?

AMD vs. CommonJS, what's the better format?

AMD Is Better For The Web Than CommonJS Modules

The Future Is Modules Not Frameworks

AMD No Longer A CommonJS Specification

On Inventing JavaScript Module Formats And Script Loaders

The AMD Mailing List

Script loaders and frameworks that support AMD

In-browser:

- RequireJS *http://requirejs.org*
- curl.js *http://github.com/unscriptable/curl*
- bdLoad *http://bdframework.com/bdLoad*
- Yabble *http://github.com/jbrantly/yabble*
- PINF *http://github.com/pinf/loader-js*
- (and more)

Server-side:

- RequireJS *http://requirejs.org*
- PINF *http://github.com/pinf/loader-js*

AMD Conclusions

Having used AMD for a number of projects, my conclusions are that it ticks a lot of the checkboxes developers creating serious applications might desire from a better module format. It avoids the need to worry about globals, supports named modules, doesn't require server transformation to function, and is a pleasure to use for dependency management.

It's also an excellent addition for modular development using Backbone.js, ember.js, or any number of other structural frameworks for keeping applications organized.

As AMD has been heavily discussed for almost two years within the Dojo and CommonJS worlds, we know it's had time to mature and evolve. We also know it's been battle-tested in the wild by a number of large companies to build nontrivial applications (IBM, BBC iPlayer), and so, if it didn't work, chances are they would have abandoned it by now, but haven't.

That said, there are still areas where AMD could be improved. Developers who have used the format for some time may feel the AMD boilerplate/wrapper code is an annoying overhead. While I share this concern, there are tools such as Volo that can help work around these issues, and I would argue that on the whole, the pros with using AMD far outweigh the cons.

CommonJS

The CommonJS module proposal specifies a simple API for declaring modules that work outside of the browser (such as on the server). Unlike AMD, it attempts to cover a broader set of concerns such as IO, filesystem, promises, and more.

Originally called ServerJS in a project started by Kevin Dangoor back in 2009, the format was more recently formalized by CommonJS, a volunteer working group that aims to design, prototype, and standardize JavaScript APIs. To date, they've attempted to ratify standards for both modules and packages.

Getting Started

From a structure perspective, a CommonJS module is a reusable piece of JavaScript that exports specific objects made available to any dependent code. Unlike AMD, there are typically no function wrappers around such modules (so we won't see `define` here, for example).

CommonJS modules basically contain two primary parts: a free variable named **`exports`**, which contains the objects a module wishes to make available to other modules, and a `require` function that modules can use to import the exports of other modules (Examples 11-9, 11-10, and 11-11).

Example 11-9. Understanding CommonJS: require() and exports

```
// package/lib is a dependency we require
var lib = require( "package/lib" );

// behaviour for our module
function foo(){
    lib.log( "hello world!" );
}

// export (expose) foo to other modules
exports.foo = foo;
```

Example 11-10. Basic consumption of exports

```
// package/lib is a dependency we require
var lib = require( "package/lib" );

// behaviour for our module
function foo(){
    lib.log( "hello world!" );
}

// export (expose) foo to other modules
exports.foo = foo;
```

Example 11-11. AMD-equivalent of the first CommonJS example

```
define(function(require){
   var lib = require( "package/lib" );

    // some behaviour for our module
    function foo(){
        lib.log( "hello world!" );
    }

    // export (expose) foo for other modules
    return {
        foobar: foo
    };
});
```

This can be done as AMD supports a simplified CommonJS wrapping feature.

Consuming Multiple Dependencies

app.js:

```
var modA = require( "./foo" );
var modB = require( "./bar" );

exports.app = function(){
    console.log( "Im an application!" );
}
```

```
exports.foo = function(){
    return modA.helloWorld();
}
```

bar.js:

```
exports.name = "bar";
```

foo.js:

```
require( "./bar" );
exports.helloWorld = function(){
    return "Hello World!!"
}
```

Loaders and Frameworks that Support CommonJS

In-browser:

- curl.js *http://github.com/unscriptable/curl*
- SproutCore 1.1 *http://sproutcore.com*
- PINF *http://github.com/pinf/loader-js*

Server-side:

- Node *http://nodejs.org*
- Narwhal *https://github.com/tlrobinson/narwhal*
- Persevere *http://www.persvr.org/*
- Wakanda *http://www.wakandasoft.com/*

Is CommonJS Suitable for the Browser?

There are developers that feel CommonJS is better suited to server-side development, which is one reason there's currently a level of *disagreement* over which format should and will be used as the de facto standard in the pre-Harmony age moving forward. Some of the arguments against CommonJS include a note that many CommonJS APIs address server-oriented features that one would simply not be able to implement at a browser level in JavaScript—for example, *io*, *system* and *js* could be considered unimplementable by the nature of their functionality.

That said, it's useful to know how to structure CommonJS modules regardless so that we can better appreciate how they fit in when defining modules that may be used everywhere. Modules that have applications on both the client and server include validation, conversion, and templating engines. The way some developers are approaching choosing which format to use is opting for CommonJS when a module can be used in a server-side environment and using AMD if this is not the case.

As AMD modules are capable of using plug-ins and can define more granular things like constructors and functions, this makes sense. CommonJS modules are only able to define objects that can be tedious to work with if we're trying to obtain constructors out of them.

Although it's beyond the scope of this section, you may have also noticed that there were different types of `require` methods mentioned when discussing AMD and CommonJS. The concern with a similar naming convention is of course confusion, and the community is currently split on the merits of a global `require` function. John Hann's suggestion here is that rather than calling it `require`, which would probably fail to achieve the goal of informing users about the different between a global and inner `require`, it may make more sense to rename the global loader method something else (e.g., the name of the library). It's for this reason that a loader like curl.js uses `curl()` as opposed to `require`.

Related Reading

Demystifying CommonJS Modules

JavaScript Growing Up

The RequireJS Notes On CommonJS

Taking Baby Steps With Node.js And CommonJS - Creating Custom Modules

Asynchronous CommonJS Modules for the Browser

The CommonJS Mailing List

AMD and CommonJS: Competing, but Equally Valid Standards

Both AMD and CommonJS are valid module formats with different end goals.

AMD adopts a browser-first approach to development, opting for asynchronous behavior and simplified backward compatibility, but it doesn't have any concept of file I/O. It supports objects, functions, constructors, strings, JSON and many other types of modules, running natively in the browser. It's incredibly flexible.

CommonJS on the other hand takes a server-first approach, assuming synchronous behavior, no global *baggage*, and attempts to cater for the future (on the server). What we mean by this is that because CommonJS supports unwrapped modules, it can feel a little more close to the ES.next/Harmony specifications, freeing us of the `define()` wrapper that AMD enforces. CommonJS modules however only support objects as modules.

UMD: AMD and CommonJS-Compatible Modules for Plug-ins

For developers wishing to create modules that can work in both browser and server-side environments, existing solutions could be considered a little lacking. To help alleviate this, James Burke, a number of other developers, and I created Universal Module Definition (UMD; *https://github.com/umdjs/umd*).

UMD is an experimental module format that allows the definition of modules that work in both client and server environments with all or most of the popular script-loading techniques available at the time of writing. Although the idea of (yet) another module format may be daunting, we will cover UMD briefly for the sake of thoroughness.

We originally began defining UMD by taking a look at the simplified CommonJS wrapper supported in the AMD specification. For developers wishing to write modules as if they were CommonJS modules, the following CommonJS-compatible format could be used:

Basic AMD hybrid format

```
define( function ( require, exports, module ){

    var shuffler = require( "lib/shuffle" );

    exports.randomize = function( input ){
        return shuffler.shuffle( input );
    }
});
```

It's important however to note that a module is really only treated as a CommonJS module if it doesn't contain a dependency array and the definition function contains one parameter at minimum. This also won't work correctly on some devices (e.g., the PS3). For further information about the above wrapper, see *http://requirejs.org/docs/api.html#cjsmodule*.

Taking this further, we wanted to provide a number of different patterns that not just worked with AMD and CommonJS, but also solved common compatibility problems developers wishing to develop such modules had with other environments.

One such variation we can see below allows us to use CommonJS, AMD, or browser globals to create a module.

Using CommonJS, AMD, or browser globals to create a module

Define a module `commonJsStrict`, which depends on another module called `b`. The name of the module is implied by the filename, and it's best practice for the file name and the exported global to have the same name.

If the module `b` also uses the same type of boilerplate in the browser, it will create a global `.b` that is used. If we don't wish to support the browser global patch, we can remove the `root` and pass `this` as the first argument to the top function.

```
(function ( root, factory ) {
    if ( typeof exports === 'object' ) {
        // CommonJS
        factory( exports, require('b') );
    } else if ( typeof define === 'function' && define.amd ) {
        // AMD. Register as an anonymous module.
        define( ['exports', 'b'], factory);
    } else {
        // Browser globals
        factory( (root.commonJsStrict = {}), root.b );
    }
}(this, function ( exports, b ) {
    //use b in some fashion.

    // attach properties to the exports object to define
    // the exported module properties.
    exports.action = function () {};
}));
```

The UMD repository contains variations covering modules that work optimally in the browser, those best for providing exports, those optimal for CommonJS runtimes, and even those that work best for defining jQuery plug-ins, which we will look at next.

jQuery plug-ins that function in all environments

UMD provides two patterns for working with jQuery plug-ins: one that defines plug-ins that work well with AMD and browser globals and another that can also work in CommonJS environments. jQuery is not likely to be used in most CommonJS environments, so keep this in mind, unless we're working with an environment that does play well with it.

We will now define a plug-in composed of a core and an extension to that core. The core plug-in is loaded into a `$.core` namespace, which can then be easily extended using plug-in extensions via the namespacing pattern. Plug-ins loaded via script tags automatically populate a `plugin` namespace under `core` (i.e., `$.core.plugin.methodName()`).

The pattern can be quite nice to work with, because plug-in extensions can access properties and methods defined in the base or, with a little tweaking, override default behavior so that it can be extended to do more. A loader is also not required to make any of this fully functional.

For more details of what is being done, please see the inline comments in the code samples below.

usage.html:

```
<script type="text/javascript" src="jquery-1.7.2.min.js"></script>
<script type="text/javascript" src="pluginCore.js"></script>
<script type="text/javascript" src="pluginExtension.js"></script>

<script type="text/javascript">

$(function(){

    // Our plug-in "core" is exposed under a core namespace in
    // this example, which we first cache
    var core = $.core;
```

```
        // Then use use some of the built-in core functionality to
        // highlight all divs in the page yellow
        core.highlightAll();

        // Access the plug-ins (extensions) loaded into the "plugin"
        // namespace of our core module:

        // Set the first div in the page to have a green background.
        core.plugin.setGreen( "div:first");
        // Here we're making use of the core's "highlight" method
        // under the hood from a plug-in loaded in after it

        // Set the last div to the "errorColor" property defined in
        // our core module/plug-in. If we review the code further down,
        // we can see how easy it is to consume properties and methods
        // between the core and other plug-ins
        core.plugin.setRed("div:last");
    });

    </script>
```

pluginCore.js:

```
// Module/plug-in core
// Note: the wrapper code we see around the module is what enables
// us to support multiple module formats and specifications by
// mapping the arguments defined to what a specific format expects
// to be present. Our actual module functionality is defined lower
// down, where a named module and exports are demonstrated.
//
// Note that dependencies can just as easily be declared if required
// and should work as demonstrated earlier with the AMD module examples.

(function ( name, definition ){
  var theModule = definition(),
      // this is considered "safe":
      hasDefine = typeof define === "function" && define.amd,
      // hasDefine = typeof define === "function",
      hasExports = typeof module !== "undefined" && module.exports;

  if ( hasDefine ){ // AMD Module
    define(theModule);
  } else if ( hasExports ) { // Node.js Module
    module.exports = theModule;
  } else { // Assign to common namespaces or simply the global object (window)
    ( this.jQuery || this.ender || this.$ || this)[name] = theModule;
  }
})( "core", function () {
    var module = this;
    module.plugins = [];
    module.highlightColor = "yellow";
    module.errorColor = "red";

  // define the core module here and return the public API
```

```
    // This is the highlight method used by the core highlightAll()
    // method and all of the plug-ins highlighting elements different
    // colors
    module.highlight = function( el,strColor ){
      if( this.jQuery ){
        jQuery(el).css( "background", strColor );
      }
    }
    return {
        highlightAll:function(){
          module.highlight("div", module.highlightColor);
        }
    };

  });
```

pluginExtension.js:

```
  // Extension to module core

  (function ( name, definition ) {
      var theModule = definition(),
          hasDefine = typeof define === "function",
          hasExports = typeof module !== "undefined" && module.exports;

      if ( hasDefine ) { // AMD Module
          define(theModule);
      } else if ( hasExports ) { // Node.js Module
          module.exports = theModule;
      } else {

          // Assign to common namespaces or simply the global object (window)
          // account for for flat-file/global module extensions
          var obj = null,
              namespaces,
              scope;

          obj = null;
          namespaces = name.split(".");
          scope = ( this.jQuery || this.ender || this.$ || this );

          for ( var i = 0; i < namespaces.length; i++ ) {
              var packageName = namespaces[i];
              if ( obj && i == namespaces.length - 1 ) {
                  obj[packageName] = theModule;
              } else if ( typeof scope[packageName] === "undefined" ) {
                  scope[packageName] = {};
              }
              obj = scope[packageName];
          }

      }
  })( "core.plugin" , function () {

      // Define our module here and return the public API.
      // This code could be easily adapted with the core to
```

```
        // allow for methods that overwrite and extend core functionality
        // in order to expand the highlight method to do more if we wish.
        return {
            setGreen: function ( el ) {
                highlight(el, "green");
            },
            setRed: function ( el ) {
                highlight(el, errorColor);
            }
        };

    });
```

UMD doesn't aim to replace AMD nor CommonJS but merely offers some supplemental assistance for developers wishing to get their code working in more environments today. For further information or to contribute suggestions toward this experimental format, see *https://github.com/umdjs/umd*.

Further reading

- Using AMD Loaders to Write and Manage Modular JavaScript, John Hann
- Demystifying CommonJS Modules, Alex Young
- AMD Module Patterns: Singleton, John Hann
- Run-Anywhere JavaScript Modules Boilerplate Code, Kris Zyp
- Standards And Proposals for JavaScript Modules And jQuery, James Burke

ES Harmony

TC39, the standards body charged with defining the syntax and semantics of ECMAScript , has been keeping a close eye on the evolution of JavaScript usage for large-scale development over the past few years. One key area they've been reviewing is the possible need to support more advanced modules that cater for the needs of the modern JavaScript developer.

For this reason, there are currently proposals for a number of exciting additions to the language, including flexible modules that can work on both the client and server, a module loader, and more. In this section, we'll explore code samples using the syntax proposed for modules in ES.next so we can get a taste of what's to come.

Although Harmony is still in the proposal phases, we can already try out (partial) features of ES.next that address native support for writing modular JavaScript, thanks to Google's Traceur compiler. To get up and running with Traceur in under a minute, read this getting started guide. There's also a JSConf presentation that's worth looking at, if you're interested in learning more about the project.

Modules with Imports and Exports

Having read through the sections on AMD and CommonJS modules, you may be familiar with the concept of module dependencies (imports) and module exports (or, the public API/variables we allow other modules to consume). In ES.next, these concepts have been proposed in a slightly more succinct manner with dependencies being specified using an `import` keyword. `export` isn't greatly different to what we might expect, and many developers will look at the code samples lower down and instantly grab them.

- `import` declarations bind a modules, exports as local variables and may be renamed to avoid name collisions/conflicts.
- `export` declarations declare that a local binding of a module is externally visible, such that other modules may read the exports but can't modify them. Interestingly, modules may export child modules but can't export modules that have been defined elsewhere. We can also rename exports so their external names differ from their local names.

```
module staff{
    // specify (public) exports that can be consumed by
    // other modules
    export var baker = {
        bake: function( item ){
            console.log( "Woo! I just baked " + item );
        }
    }
}

module skills{
    export var specialty = "baking";
    export var experience = "5 years";
}

module cakeFactory{

    // specify dependencies
    import baker from staff;

    // import everything with wildcards
    import * from skills;

    export var oven = {
        makeCupcake: function( toppings ){
            baker.bake( "cupcake", toppings );
        },
        makeMuffin: function( mSize ){
            baker.bake( "muffin", size );
        }
    }
}
```

Modules Loaded from Remote Sources

The module proposals also cater for modules which are remotely based (e.g. a third-party libraries) making it simplistic to load modules in from external locations. Here's an example of pulling in the module we defined above and utilizing it:

```
module cakeFactory from "http://addyosmani.com/factory/cakes.js";
cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

Module Loader API

The module loader proposed describes a dynamic API for loading modules in highly controlled contexts. Signatures supported on the loader include `load(url, moduleInstance, error)` for loading modules, `createModule(object, globalModuleReferences)`, and others.

Here's another example for dynamically loading in the module we initially defined. Note that unlike the last example where we pulled in a module from a remote source, the module loader API is better suited to dynamic contexts.

```
Loader.load( "http://addyosmani.com/factory/cakes.js" ,
    function( cakeFactory ){
        cakeFactory.oven.makeCupcake( "chocolate" );
    });
```

CommonJS-like Modules for the Server

For developers who are more interested in server environments, the module system proposed for ES.next isn't just constrained to looking at modules in the browser. Here, for example, we can see a CommonJS-like module proposed for use on the server:

```
// io/File.js
export function open( path ) { ... };
export function close( hnd ) { ... };

// compiler/LexicalHandler.js
module file from "io/File";

import { open, close } from file;
export function scan( in ) {
    try {
        var h = open( in ) ...
    }
    finally { close( h ) }
}

module lexer from "compiler/LexicalHandler";
module stdlib from "@std";

//... scan(cmdline[0]) ...
```

Classes with Constructors, Getters, and Setters

The notion of a class has always been a contentious issue with purists, and we've so far got along with either falling back on JavaScript's prototypal nature or through using frameworks or abstractions that offer the ability to use *class* definitions in a form that de-sugars to the same prototypal behavior.

In Harmony, classes have been proposed for the language along with constructors and (finally) some sense of true privacy. In the following examples, inline comments are provided to help explain how classes are structured.

Reading through, one may also notice the lack of the word "function" in here. This isn't a typo error: TC39 have been making a conscious effort to decrease our abuse of the `function` keyword for everything, and the hope is that this will help simplify how we write code.

```
class Cake{

    // We can define the body of a class" constructor
    // function by using the keyword "constructor" followed
    // by an argument list of public and private declarations.
    constructor( name, toppings, price, cakeSize ){
        public name = name;
        public cakeSize = cakeSize;
        public toppings = toppings;
        private price = price;

    }

    // As a part of ES.next's efforts to decrease the unnecessary
    // use of "function" for everything, you'll notice that it's
    // dropped for cases such as the following. Here an identifier
    // followed by an argument list and a body defines a new method

    addTopping( topping ){
        public( this ).toppings.push( topping );
    }

    // Getters can be defined by declaring get before
    // an identifier/method name and a curly body.
    get allToppings(){
        return public( this ).toppings;
    }

    get qualifiesForDiscount(){
        return private( this ).price > 5;
    }

    // Similar to getters, setters can be defined by using
    // the "set" keyword before an identifier
    set cakeSize( cSize ){
        if( cSize < 0 ){
            throw new Error( "Cake must be a valid size -
            either small, medium or large" );
        }
        public( this ).cakeSize = cSize;
    }


}
```

ES Harmony Conclusions

As we've seen, Harmony might come with some exciting new additions that will ease the development of modular applications and handling concerns such as dependency management.

At present, our best options for using Harmony syntax in today's browsers is through a transpiler such as Google Traceur or Esprima. There are also projects such as Require HM that allow us to use Harmony modules with AMD. Our best bets however until we have specification finalization are AMD (for in-browser modules) and CommonJS (for those on the server).

Related Reading

A First Look At The Upcoming JavaScript Modules

David Herman On JavaScript/ES.Next (Video)

ES Harmony Module Proposals

ES Harmony Module Semantics/Structure Rationale

ES Harmony Class Proposals

Conclusions

In this section we reviewed several of the options available for writing modular JavaScript using modern module formats.

These formats have a number of advantages over using the module pattern alone including: avoiding the need to manage global variables, better support for static and dynamic dependency management, improved compatibility with script loaders, better compatibility for modules on the server, and more.

In short, I recommend trying out what's been suggested in this chapter, as these formats offer a great deal of power and flexibility that can significantly assist with better organizing our applications.

CHAPTER 12

Design Patterns in jQuery

jQuery is currently the most popular JavaScript DOM manipulation library and provides an abstracted layer for interacting with the DOM in a safe, cross-browser manner. Interestingly, the library also serves as an example of how design patterns can be effectively used to create an API that is both readable and easy to use.

While in many cases the core contributors that wrote jQuery didn't set out to use specific patterns, they exist there regardless and are useful to learn from. Let's take a look at what some of these patterns are and how they are used in the API.

The Composite Pattern

The *Composite pattern* describes a group of objects that can be treated in the same way a single instance of an object may be.

This allows us to treat both individual objects and compositions in a uniform manner, meaning that the same behavior will be applied regardless of whether we're working with one item or a thousand.

In jQuery, when we're applying methods to an element or collection of elements, we can treat both sets in a uniform manner, as both selections return a jQuery object.

This is demonstrated by the code sample using the jQuery selector below. Here, it's possible to add an `active` class to both selections for a single element (e.g., an element with a unique ID) or a group of elements with the same tag name or class, without additional effort:

```
// Single elements
$( "#singleItem" ).addClass( "active" );
$( "#container" ).addClass( "active" );

// Collections of elements
$( "div" ).addClass( "active" );
$( ".item" ).addClass( "active" );
$( "input" ).addClass( "active" );
```

The jQuery `addClass()` implementation could either directly use native `for` loops (or jQuery's `jQuery.each()`/`jQuery.fn.each()`) to iterate through a collection in order to apply the method to both single items or groups. Looking through the source, we can see this is indeed the case:

```
    addClass: function( value ) {
    var classNames, i, l, elem,
      setClass, c, cl;

    if ( jQuery.isFunction( value ) ) {
      return this.each(function( j ) {
        jQuery( this ).addClass( value.call(this, j, this.className) );
      });
    }

    if ( value && typeof value === "string" ) {
      classNames = value.split( rspace );

      for ( i = 0, l = this.length; i < l; i++ ) {
        elem = this[ i ];

        if ( elem.nodeType === 1 ) {
          if ( !elem.className && classNames.length === 1 ) {
            elem.className = value;

          } else {
            setClass = " " + elem.className + " ";

            for ( c = 0, cl = classNames.length; c < cl; c++ ) {
              if ( !~setClass.indexOf( " " + classNames[ c ] + " " ) ) {
                setClass += classNames[ c ] + " ";
              }
            }
            elem.className = jQuery.trim( setClass );
          }
        }
      }
    }

    return this;
  }
```

The Adapter Pattern

The *Adapter pattern* translates an *interface* for an object or class into an interface compatible with a specific system.

Adapters basically allow objects or classes to function together, which normally they couldn't due to their incompatible interfaces. The adapter translates calls to its interface into calls to the original interface, and the code required to achieve this is usually quite minimal.

One example of an adapter we may have used is the jQuery `jQuery.fn.css()` method. It helps normalize the interfaces to show how styles can be applied across a number of browsers, making in trivial for us to use a simple syntax that is adapted to use what the browser actually supports behind the scenes:

```
// Cross browser opacity:
// opacity: 0.9;  Chrome 4+, FF2+, Saf3.1+, Opera 9+, IE9, iOS 3.2+, Android 2.1+
// filter: alpha(opacity=90);  IE6-IE8

// Setting opacity
$( ".container" ).css( { opacity: .5 } );

// Getting opacity
var currentOpacity = $( ".container" ).css('opacity');
```

The corresponding jQuery core cssHook, which makes the above possible, can be seen below:

```
get: function( elem, computed ) {
  // IE uses filters for opacity
  return ropacity.test( (
        computed && elem.currentStyle ?
            elem.currentStyle.filter : elem.style.filter) || "" ) ?
    ( parseFloat( RegExp.$1 ) / 100 ) + "" :
    computed ? "1" : "";
},

set: function( elem, value ) {
  var style = elem.style,
    currentStyle = elem.currentStyle,
    opacity = jQuery.isNumeric( value ) ?
          "alpha(opacity=" + value * 100 + ")" : "",
    filter = currentStyle && currentStyle.filter || style.filter || "";

  // IE has trouble with opacity if it does not have layout
  // Force it by setting the zoom level
  style.zoom = 1;

  // if setting opacity to 1, and no other filters
  //exist - attempt to remove filter attribute #6652
  if ( value >= 1 && jQuery.trim( filter.replace( ralpha, "" ) ) === "" ) {

    // Setting style.filter to null, "" & " " still leave
    // "filter:" in the cssText if "filter:" is present at all,
    // clearType is disabled, we want to avoid this style.removeAttribute
    // is IE Only, but so apparently is this code path...
    style.removeAttribute( "filter" );

    // if there there is no filter style applied in a css rule, we are done
    if ( currentStyle && !currentStyle.filter ) {
      return;
    }
  }

  // otherwise, set new filter values
```

```
      style.filter = ralpha.test( filter ) ?
        filter.replace( ralpha, opacity ) :
        filter + " " + opacity;
    }
  };
```

The Facade Pattern

As we reviewed earlier in the book, the *Facade pattern* provides a simpler abstracted interface to a larger (potentially more complex) body of code.

Facades can be frequently found across the jQuery library and provide developers easy access to implementations for handling DOM manipulation, animation, and of particular interest, cross-browser Ajax.

The following are facades for jQuery's `$.ajax()`:

```
$.get( url, data, callback, dataType );
$.post( url, data, callback, dataType );
$.getJSON( url, data, callback );
$.getScript( url, callback );
```

These are translated behind the scenes to:

```
// $.get()
$.ajax({
  url: url,
  data: data,
  dataType: dataType
}).done( callback );

// $.post
$.ajax({
  type: "POST",
  url: url,
  data: data,
  dataType: dataType
}).done( callback );

// $.getJSON()
$.ajax({
  url: url,
  dataType: "json",
  data: data,
}).done( callback );

// $.getScript()
$.ajax({
  url: url,
  dataType: "script",
}).done( callback );
```

What's even more interesting is that the above facades are actually facades in their own right, hiding a great deal of complexity behind the scenes.

This is because the `jQuery.ajax()` implementation in jQuery core is a nontrivial piece of code to say the least. At minimum, it normalizes the cross-browser differences between XHR (`XMLHttpRequest`) and makes it trivial for us to perform common HTTP actions (e.g `get`, `post`, etc.,), work with Deferreds, and so on.

As it would take an entire chapter to show all of the code related to the above facades, here is instead the code in jQuery core normalizing XHR:

```
// Functions to create xhrs
function createStandardXHR() {
  try {
    return new window.XMLHttpRequest();
  } catch( e ) {}
}

function createActiveXHR() {
  try {
    return new window.ActiveXObject( "Microsoft.XMLHTTP" );
  } catch( e ) {}
}

// Create the request object
jQuery.ajaxSettings.xhr = window.ActiveXObject ?
  /* Microsoft failed to properly
   * implement the XMLHttpRequest in IE7 (can't request local files),
   * so we use the ActiveXObject when it is available
   * Additionally XMLHttpRequest can be disabled in IE7/IE8 so
   * we need a fallback.
   */
  function() {
    return !this.isLocal && createStandardXHR() || createActiveXHR();
  } :
  // For all other browsers, use the standard XMLHttpRequest object
  createStandardXHR;
  ...
```

While the following block of code is also a level above the actual jQuery XHR (`jqXHR`) implementation, it's the convenience facade that we actually most commonly interact with:

```
// Request the remote document
jQuery.ajax({
  url: url,
  type: type,
  dataType: "html",
  data: params,
  // Complete callback (responseText is used internally)
  complete: function( jqXHR, status, responseText ) {
    // Store the response as specified by the jqXHR object
    responseText = jqXHR.responseText;
    // If successful, inject the HTML into all the matched elements
    if ( jqXHR.isResolved() ) {
      // Get the actual response in case
      // a dataFilter is present in ajaxSettings
      jqXHR.done(function( r ) {
```

```
                responseText = r;
              });
              // See if a selector was specified
              self.html( selector ?
                // Create a dummy div to hold the results
                jQuery("")
                  // inject the contents of the document in, removing the scripts
                  // to avoid any 'Permission Denied' errors in IE
                  .append(responseText.replace(rscript, ""))

                  // Locate the specified elements
                  .find(selector) :

                // If not, just inject the full result
                responseText );
            }

            if ( callback ) {
              self.each( callback, [ responseText, status, jqXHR ] );
            }
          }
        });

        return this;
      }
```

The Observer Pattern

Another pattern we reviewed earlier is the Observer (Publish/Subscribe) pattern. This is where the objects in a system may subscribe to other objects and be notified by them when an event of interest occurs.

jQuery core has come with built-in support for a Publish/Subscribe-like system for a few years now, which it refers to as *custom events*.

In earlier versions of the library, access to these custom events was possible using `jQuery.bind()` (subscribe), `jQuery.trigger()` (publish), and `jQuery.unbind()` (unsubscribe), but in recent versions, this can be done using `jQuery.on()`, `jQuery.trigger()`, and `jQuery.off()`.

Here we can see an example of this being used in practice:

```
// Equivalent to subscribe(topicName, callback)
$( document ).on( "topicName" , function () {
    //..perform some behaviour
});

// Equivalent to publish(topicName)
$( document ).trigger( "topicName" );

// Equivalent to unsubscribe(topicName)
$( document ).off( "topicName" );
```

Calls to `jQuery.on()` and `jQuery.off()` eventually go through the jQuery events system. Similar to Ajax, as the implementation for this is relatively long, we can instead look at where and how the actual event handlers for custom events are attached:

```
jQuery.event = {

  add: function( elem, types, handler, data, selector ) {

    var elemData, eventHandle, events,
      t, tns, type, namespaces, handleObj,
      handleObjIn, quick, handlers, special;

    ...

    // Init the element's event structure and main handler,
    //if this is the first
    events = elemData.events;
    if ( !events ) {
      elemData.events = events = {};
    }
    ...

    // Handle multiple events separated by a space
    // jQuery(...).bind("mouseover mouseout", fn);
    types = jQuery.trim( hoverHack(types) ).split( " " );
    for ( t = 0; t < types.length; t++ ) {

      ...

      // Init the event handler queue if we're the first
      handlers = events[ type ];
      if ( !handlers ) {
        handlers = events[ type ] = [];
        handlers.delegateCount = 0;

        // Only use addEventListener/attachEvent if the special
        // events handler returns false
        if ( !special.setup || special.setup.call( elem, data,
        //namespaces, eventHandle ) === false ) {
          // Bind the global event handler to the element
          if ( elem.addEventListener ) {
            elem.addEventListener( type, eventHandle, false );

          } else if ( elem.attachEvent ) {
            elem.attachEvent( "on" + type, eventHandle );
          }
        }
      }
```

For those that prefer to use the conventional naming scheme for the Observer pattern, Ben Alman created a simple wrapper around the above methods that provides us access to the `jQuery.publish()`, `jQuery.subscribe`, and `jQuery.unsubscribe` methods. I've previously linked to them earlier in the book, but we can see the wrapper in full below.

```
(function( $ ) {

  var o = $({});

  $.subscribe = function() {
    o.on.apply(o, arguments);
  };

  $.unsubscribe = function() {
    o.off.apply(o, arguments);
  };

  $.publish = function() {
    o.trigger.apply(o, arguments);
  };

}( jQuery ));
```

In recent versions of jQuery, a multipurpose callbacks object (`jQuery.Callbacks`) was made available to enable users to write new solutions based on callback lists. One such solution to write using this feature is another Publish/Subscribe system. An implementation of this is the following:

```
var topics = {};

jQuery.Topic = function( id ) {
    var callbacks,
        topic = id && topics[ id ];
    if ( !topic ) {
        callbacks = jQuery.Callbacks();
        topic = {
            publish: callbacks.fire,
            subscribe: callbacks.add,
            unsubscribe: callbacks.remove
        };
        if ( id ) {
            topics[ id ] = topic;
        }
    }
    return topic;
};
```

which can then be used as follows:

```
// Subscribers
$.Topic( "mailArrived" ).subscribe( fn1 );
$.Topic( "mailArrived" ).subscribe( fn2 );
$.Topic( "mailSent" ).subscribe( fn1 );

// Publisher
$.Topic( "mailArrived" ).publish( "hello world!" );
$.Topic( "mailSent" ).publish( "woo! mail!" );

//  Here, "hello world!" gets pushed to fn1 and fn2
//  when the "mailArrived" notification is published
//  with "woo! mail!" also being pushed to fn1 when
//  the "mailSent" notification is published.

// Outputs:
// hello world!
// fn2 says: hello world!
// woo! mail!
```

The Iterator Pattern

The Iterator is a design pattern in which iterators (objects that allow us to traverse through all the elements of a collection) access the elements of an aggregate object sequentially without needing to expose its underlying form.

Iterators encapsulate the internal structure of how that particular iteration occurs. In the case of jQuery's `jQuery.fn.each()` iterator, we are actually able to use the underlying code behind `jQuery.each()` to iterate through a collection, without needing to see or understand the code working behind the scenes providing this capability.

This pattern could be considered a special case of the facade, where we explicitly deal with problems related to iteration.

```
$.each( ["john","dave","rick","julian"] , function( index, value ) {
  console.log( index + ": "" + value);
});

$( "li" ).each( function ( index ) {
  console.log( index + ": " + $( this ).text());
});
```

Here we can see the code for `jQuery.fn.each()`:

```
// Execute a callback for every element in the matched set.
each: function( callback, args ) {
  return jQuery.each( this, callback, args );
}
```

Followed by the code behind `jQuery.each()`, which handles two ways of iterating through objects:

```
  each: function( object, callback, args ) {
    var name, i = 0,
      length = object.length,
      isObj = length === undefined || jQuery.isFunction( object );

    if ( args ) {
      if ( isObj ) {
        for ( name in object ) {
          if ( callback.apply( object[ name ], args ) === false ) {
            break;
          }
        }
      } else {
        for ( ; i < length; ) {
          if ( callback.apply( object[ i++ ], args ) === false ) {
```

```
          break;
        }
      }
    }

  // A special, fast, case for the most common use of each
  } else {
    if ( isObj ) {
      for ( name in object ) {
        if ( callback.call( object[ name ], name, object[ name ] ) === false ) {
          break;
        }
      }
    } else {
      for ( ; i < length; ) {
        if ( callback.call( object[ i ], i, object[ i++ ] ) === false ) {
          break;
        }
      }
    }
  }

  return object;
};
```

Lazy Initialization

Lazy initialization is a design pattern that allows us to delay expensive processes until the first instance they are needed. An example of this is the `.ready()` function in jQuery that only executes a callback once the DOM is ready.

```
$( document ).ready( function () {

    // The ajax request won't attempt to execute until
    // the DOM is ready

    var jqxhr = $.ajax({
      url: "http://domain.com/api/",
      data: "display=latest&order=ascending"
    })
    .done( function( data ) ){
        $(".status").html( "content loaded" );
        console.log( "Data output:" + data );
    });

});
```

`jQuery.fn.ready()` is powered by `jQuery.bindReady()`, seen below:

```
bindReady: function() {
  if ( readyList ) {
    return;
  }
```

```
        readyList = jQuery.Callbacks( "once memory" );

        // Catch cases where $(document).ready() is called after the
        // browser event has already occurred.
        if ( document.readyState === "complete" ) {
          // Handle it asynchronously to allow scripts the opportunity to delay ready
          return setTimeout( jQuery.ready, 1 );
        }

        // Mozilla, Opera and webkit support this event
        if ( document.addEventListener ) {
          // Use the handy event callback
          document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );

          // A fallback to window.onload, that will always work
          window.addEventListener( "load", jQuery.ready, false );

        // If IE event model is used
        } else if ( document.attachEvent ) {
          // ensure firing before onload,
          // maybe late but safe also for iframes
          document.attachEvent( "onreadystatechange", DOMContentLoaded );

          // A fallback to window.onload, that will always work
          window.attachEvent( "onload", jQuery.ready );

          // If IE and not a frame
          // continually check to see if the document is ready
          var toplevel = false;

          try {
            toplevel = window.frameElement == null;
          } catch(e) {}

          if ( document.documentElement.doScroll && toplevel ) {
            doScrollCheck();
          }
        }
      },
```

While not directly used in jQuery core, some developers may also be familiar with the concept of LazyLoading via plug-ins such as this.

LazyLoading is effectively the same as Lazy initialization and is a technique whereby additional data on a page is loaded when needed (e.g., when a user has scrolled to the end of the page). In recent years, this pattern has become quite prominent and can be currently be found in both the Twitter and Facebook UIs.

The Proxy Pattern

There are times when it is necessary for us to control the access and context behind an object, and this is where the Proxy pattern can be useful.

It can help us control when an expensive object should be instantiated, provide advanced ways to reference the object, or modify the object to function a particular way in specific contexts.

In jQuery core, a `jQuery.proxy()` method exists that accepts as input a function and returns a new one that will always have a specific context. This ensures that the value of `this` within a function is the value we expect.

An example of where this is useful is when we're making use of timers within a `click` event handler. Imagine we have the following handler prior to adding any timers:

```
$( "button" ).on( "click", function () {
  // Within this function, "this" refers to the element that was clicked
  $( this ).addClass( "active" );
});
```

If we wished to add a hard delay before the `active` class was added, we *could* use `setTimeout()` to achieve this. Unfortunately there is a small problem with this solution: whatever function is passed to `setTimeout()` will have a different value for `this` within that function. It will instead refer to the `window` object, which is not what we desire.

```
$( "button" ).on( "click", function () {
  setTimeout(function () {
    // "this" doesn't refer to our element!
    // It refers to window
    $( this ).addClass( "active" );
  });
});
```

To work around this problem, we can use `jQuery.proxy()` to implement a type of proxy pattern. By calling it with the function and value we would like assigned to `this`, it will actually return a function that retains the value we desire within the correct context. Here's how this would look:

```
$( "button" ).on( "click", function () {

    setTimeout( $.proxy( function () {
        // "this" now refers to our element as we wanted
        $( this ).addClass( "active" );
    }, this), 500);

    // the last "this" we're passing tells $.proxy() that our DOM element
    // is the value we want "this" to refer to.
});
```

jQuery's implementation of `jQuery.proxy()` can be found below:

```
  // Bind a function to a context, optionally partially applying any
  // arguments.
  proxy: function( fn, context ) {
    if ( typeof context === "string" ) {
      var tmp = fn[ context ];
      context = fn;
      fn = tmp;
    }
    // Quick check to determine if target is callable, in the spec
    // this throws a TypeError, but we will just return undefined.
    if ( !jQuery.isFunction( fn ) ) {
      return undefined;
    }

    // Simulated bind
    var args = slice.call( arguments, 2 ),
      proxy = function() {
        return fn.apply( context, args.concat( slice.call( arguments ) ) );
      };

    // Set the guid of unique handler to the same of original handler,
    so it can be removed
    proxy.guid = fn.guid = fn.guid || proxy.guid || jQuery.guid++;
    return proxy;
  }
```

The Builder Pattern

When working with the DOM, we often want to construct new elements dynamically—a process that can increase in complexity depending on the final markup, attributes, and properties we wish our constructed elements to contain.

Complex elements require special care when being defined, especially if we want the flexibility to either literally define the final markup for our elements (which can get messy) or take a more readable object-oriented route instead. Having a mechanism for building our complex DOM objects that is independent from the objects themselves gives us this flexibility, and that is exactly what the Builder pattern provides.

Builders allow us to construct complex objects by only specifying the type and content of the object, shielding us from the process of creating or representing the object explicitly.

The jQuery dollar sign allows us to do just this, as it provides a number of different means for dynamically building new jQuery (and DOM) objects, by either passing in the complete markup for an element, partial markup and content, or using the jQuery for construction:

```
$( "<div class= "foo">bar</div>" );

$( "<p id="test">foo <em>bar</em></p>").appendTo("body" );

var newParagraph = $( "<p />" ).text( "Hello world" );

$( "<input />" )
      .attr({ "type": "text", "id":"sample"});
      .appendTo("#container");
```

Below is a snippet from jQuery core's internal `jQuery.prototype` method, which assists with the construction of jQuery objects from markup passed to the `jQuery()` selector.

Regardless of whether or not `document.createElement` is used to create a new element, a reference to the element (found or created) is injected into the returned object so further methods such as `.attr()` can be easily used on it right after.

```
    // HANDLE: $(html) -> $(array)
      if ( match[1] ) {
        context = context instanceof jQuery ? context[0] : context;
        doc = ( context ? context.ownerDocument || context : document );

        // If a single string is passed in and it's a single tag
        // just do a createElement and skip the rest
        ret = rsingleTag.exec( selector );

        if ( ret ) {
          if ( jQuery.isPlainObject( context ) ) {
            selector = [ document.createElement( ret[1] ) ];
            jQuery.fn.attr.call( selector, context, true );

          } else {
            selector = [ doc.createElement( ret[1] ) ];
          }

        } else {
          ret = jQuery.buildFragment( [ match[1] ], [ doc ] );
          selector = ( ret.cacheable ? jQuery.clone(ret.fragment)
          : ret.fragment ).childNodes;
        }

        return jQuery.merge( this, selector );
```

CHAPTER 13

jQuery Plug-in Design Patterns

jQuery plug-in development has evolved over the past few years. We no longer have just one way to write plug-ins, but many. In reality, certain plug-in design patterns might work better for a particular problem or component than others.

Some developers may wish to use the jQuery UI widget factory; it's great for complex, flexible UI components. Some may not.

Some might like to structure their plug-ins more like modules (similar to the module pattern) or use a more modern module format such as AMD.

Some might want their plug-ins to harness the power of prototypal inheritance. Others may wish to use custom events or Publish/Subscribe to communicate from plug-ins to the rest of their app. And so on.

I began to think about plug-in patterns after noticing a number of efforts to create a one-size-fits-all jQuery plug-in boilerplate. While such a boilerplate is a great idea in theory, the reality is that we rarely write plug-ins in one fixed way, using a single pattern all the time.

Let us assume that we've tried our hand at writing our own jQuery plug-ins at some point and we're comfortable putting together something that works. It's functional. It does what it needs to do, but perhaps we feel it could be structured better. Maybe it could be more flexible or could be designed to address more of the issues developers commonly run into. If this sounds familiar, then you might find this chapter useful. In it, we're going to explore a number of jQuery plug-in patterns that have worked well for other developers in the wild.

Note: This chapter is targeted at intermediate to advanced developers, although we will briefly review some jQuery plug-in fundamentals to begin.

If you don't feel quite ready for this just yet, I'm happy to recommend the official jQuery Plug-ins/Authoring guide, Ben Alman's plug-in style guide, and Remy Sharp's "Signs of a Poorly Written jQuery Plug-in" as reading material prior to starting this section.

Patterns

jQuery plug-ins have few concrete rules, which is one of the reasons for the incredible diversity in how they are implemented across the community. At the most basic level, we can write a plug-in simply by adding a new function property to jQuery's `jQuery.fn` object, as follows:

```
$.fn.myPluginName = function () {
    // our plugin logic
};
```

This is great for compactness, but the following would be a better foundation to build on:

```
(function( $ ){
  $.fn.myPluginName = function () {
    // our plugin logic
  };
})( jQuery );
```

Here, we've wrapped our plug-in logic in an anonymous function. To ensure that our use of the `$` sign as a shorthand creates no conflicts between jQuery and other JavaScript libraries, we simply pass it to this closure, which maps it to the dollar sign. This ensures that it can't be affected by anything outside of its scope of execution.

An alternative way to write this pattern would be to use `jQuery.extend()`, which enables us to define multiple functions at once and sometimes make more sense semantically:

```
(function( $ ){
    $.extend($.fn, {
        myplugin: function(){
            // your plugin logic
        }
    });
})( jQuery );
```

We have now reviewed some jQuery plug-in fundamentals, but a lot more could be done to take this further. *A Lightweight Start* is the first complete plug-in design pattern we'll be exploring, and it covers some best practices that we can use for basic everyday plug-in development, taking into account common gotchas worth applying.

While most of the patterns below will be explained, I recommend reading through the comments in the code, because they will offer more insight into why certain best practices are applied.

I should also mention that none of this would be possible without the previous work, input and advice of other members of the jQuery community. I've listed them inline with each pattern so you can read up on their individual work if interested.

A Lightweight Start Pattern

Let's begin our deeper look at plug-in patterns with something basic that follows best practices (including those in the jQuery plug-in authoring guide). This pattern is ideal for developers who are either new to plug-in development or who just want to achieve something simple (such as a utility plug-in). A *Lightweight start* uses the following:

- Common best practices such as a semicolon placed before the function's invocation (we'll go through why in the comments below).
- `window`, `document`, and `undefined` passed in as arguments.
- A basic defaults object.
- A simple plug-in constructor for logic related to the initial creation and the assignment of the element to work with.
- Extending the options with defaults.
- A lightweight wrapper around the constructor, which helps to avoid issues such as multiple instantiations.
- Adherence to the jQuery core style guidelines for maximized readability.

```
/*!
 * jQuery lightweight plugin boilerplate
 * Original author: @ajpiano
 * Further changes, comments: @addyosmani
 * Licensed under the MIT license
 */


// the semi-colon before the function invocation is a safety
// net against concatenated scripts and/or other plug-ins
// that are not closed properly.
;(function ( $, window, document, undefined ) {

    // undefined is used here as the undefined global
    // variable in ECMAScript 3 and is mutable (i.e. it can
    // be changed by someone else). undefined isn't really
    // being passed in so we can ensure that its value is
    // truly undefined. In ES5, undefined can no longer be
    // modified.

    // window and document are passed through as local
    // variables rather than as globals, because this (slightly)
    // quickens the resolution process and can be more
    // efficiently minified (especially when both are
    // regularly referenced in our plug-in).

    // Create the defaults once
    var pluginName = "defaultPluginName",
        defaults = {
            propertyName: "value"
        };
    // The actual plug-in constructor
    function Plugin( element, options ) {
        this.element = element;

        // jQuery has an extend method that merges the
        // contents of two or more objects, storing the
        // result in the first object. The first object
        // is generally empty because we don't want to alter
        // the default options for future instances of the plug-in
        this.options = $.extend( {}, defaults, options) ;

        this._defaults = defaults;
        this._name = pluginName;

        this.init();
    }

    Plugin.prototype.init = function () {
        // Place initialization logic here
        // We already have access to the DOM element and
        // the options via the instance, e.g. this.element
        // and this.options
    };

    // A really lightweight plug-in wrapper around the constructor,
    // preventing against multiple instantiations
    $.fn[pluginName] = function ( options ) {
        return this.each(function () {
            if ( !$.data(this, "plugin_" + pluginName )) {
                $.data( this, "plugin_" + pluginName,
                new Plugin( this, options ));
            }
        });
    }

})( jQuery, window, document );
```

Usage:

```
    $("#elem").defaultPluginName({
      propertyName: "a custom value"
    });
```

Further Reading

- Plug-ins/Authoring, jQuery
- Signs of a Poorly Written jQuery Plug-in, Remy Sharp
- How to Create Your Own jQuery Plug-in, Elijah Manor
- Style in jQuery Plug-ins and Why It Matters, Ben Almon
- Create Your First jQuery Plug-in, Part 2, Andrew Wirick

Complete Widget Factory Pattern

While the jQuery plug-in authoring guide is a great introduction to plug-in development, it doesn't help obscure away common plug-in plumbing tasks that we have to deal with on a regular basis.

The jQuery UI Widget Factory is a solution to this problem that helps us build complex, stateful plug-ins based on object-oriented principles. It also eases communication with our plug-ins instance, obfuscating a number of the repetitive tasks that we would have to code when working with basic plug-ins.

Stateful plug-ins help us keep track of their current state, also allowing us to change properties of the plug-in after it has been initialized.

One of the great things about the Widget Factory is that the majority of the jQuery UI library actually uses it as a base for its components. This means that if we're looking for further guidance on structure beyond this pattern, we won't have to look beyond the jQuery UI repository on GitHub (*https://github.com/jquery/jquery-ui*).

This jQuery UI Widget Factory pattern covers almost all of the supported default factory methods, including triggering events. As per the last pattern, comments are included for all of the methods used, and further guidance is given in the inline comments.

```
/*!
 * jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)
 * Author: @addyosmani
 * Further changes: @peolanha
 * Licensed under the MIT license
 */


;(function ( $, window, document, undefined ) {

    // define our widget under a namespace of your choice
    // with additional parameters e.g.
    // $.widget( "namespace.widgetname", (optional) - an
    // existing widget prototype to inherit from, an object
    // literal to become the widget's prototype );

    $.widget( "namespace.widgetname" , {

        //Options to be used as defaults
        options: {
            someValue: null
        },

        //Setup widget (e.g. element creation, apply theming
        // , bind events etc.)
        _create: function () {

            // _create will automatically run the first time
            // this widget is called. Put the initial widget
            // setup code here, then we can access the element
            // on which the widget was called via this.element.
            // The options defined above can be accessed
            // via this.options this.element.addStuff();
        },

        // Destroy an instantiated plug-in and clean up
        // modifications the widget has made to the DOM
        destroy: function () {

            // this.element.removeStuff();
            // For UI 1.8, destroy must be invoked from the
            // base widget
            $.Widget.prototype.destroy.call( this );
            // For UI 1.9, define _destroy instead and don't
            // worry about
            // calling the base widget
        },

        methodB: function ( event ) {
            //_trigger dispatches callbacks the plug-in user
            // can subscribe to
            // signature: _trigger( "callbackName" , [eventObject],
            // [uiObject] )
            // e.g. this._trigger( "hover", e /*where e.type ==
            // "mouseenter"*/, { hovered: $(e.target)});
            this._trigger( "methodA", event, {
                key: value
            });
        },

        methodA: function ( event ) {
            this._trigger( "dataChanged", event, {
                key: value
            });
        },

        // Respond to any changes the user makes to the
        // option method
        _setOption: function ( key, value ) {
            switch ( key ) {
            case "someValue":
                // this.options.someValue = doSomethingWith( value );
                break;
            default:
                // this.options[ key ] = value;
                break;
            }

            // For UI 1.8, _setOption must be manually invoked
            // from the base widget
            $.Widget.prototype._setOption.apply( this, arguments );
            // For UI 1.9 the _super method can be used instead
            // this._super( "_setOption", key, value );
        }
    });
```

```
})( jQuery, window, document );
```

Usage:

```
var collection = $("#elem").widgetName({
  foo: false
});

collection.widgetName("methodB");
```

Further Reading

- The jQuery UI Widget Factory
- Introduction to Stateful Plug-ins and the Widget Factory, Doug Neiner
- Widget Factory (explained), Scott Gonzalez
- Understanding jQuery UI Widgets: A Tutorial, Hacking at 0300

Nested Namespacing Plug-in Pattern

As we've previously covered in the book, namespacing our code is a way to avoid collisions with other objects and variables in the global namespace. They're important because we want to safeguard our plug-in from breaking in the event that another script on the page uses the same variable or plug-in names as ours. As a good citizen of the global namespace, we must also do our best not to prevent other developers scripts from executing because of the same issues.

JavaScript doesn't really have built-in support for namespaces as other languages do, but it does have objects that can be used to achieve a similar effect. Employing a top-level object as the name of our namespace, we can easily check for the existence of another object on the page with the same name. If such an object does not exist, then we define it; if it does exist, then we simply extend it with our plug-in.

Objects (or, rather, object literals) can be used to create nested namespaces, such as `namespace.subnamespace.pluginName` and so on. But to keep things simple, the namespacing boilerplate below should offer us everything we need to get started with these concepts.

```
/*!
 * jQuery namespaced "Starter" plugin boilerplate
 * Author: @dougneiner
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

;(function ( $ ) {
    if (!$.myNamespace) {
        $.myNamespace = {};
    };
```

```
        $.myNamespace.myPluginName = function ( el, myFunctionParam, options ) {
            // To avoid scope issues, use "base" instead of "this"
            // to reference this class from internal events and functions.
            var base = this;

            // Access to jQuery and DOM versions of element
            base.$el = $( el );
            base.el = el;

            // Add a reverse reference to the DOM object
            base.$el.data( "myNamespace.myPluginName" , base );

            base.init = function () {
                base.myFunctionParam = myFunctionParam;

                base.options = $.extend({},
                $.myNamespace.myPluginName.defaultOptions, options);

                // Put our initialization code here
            };

            // Sample Function, Uncomment to use
            // base.functionName = function( parameters ){
            //
            // };
            // Run initializer
            base.init();
        };

        $.myNamespace.myPluginName.defaultOptions = {
            myDefaultValue: ""
        };

        $.fn.mynamespace_myPluginName = function
            ( myFunctionParam, options ) {
            return this.each(function () {
                (new $.myNamespace.myPluginName( this,
                myFunctionParam, options ));
            });
        };

    })( jQuery );
```

Usage:

```
    $("#elem").mynamespace_myPluginName({
      myDefaultValue: "foobar"
    });
```

Further Reading

- Namespacing in JavaScript, Angus Croll
- Use Your $.fn jQuery Namespace, Ryan Florence

- JavaScript Namespacing, Peter Michaux
- Modules and namespaces in JavaScript, Axel Rauschmayer

Custom Events Plug-in Pattern (with the Widget Factory)

In *Chapter 9* of this book, we discussed the Observer pattern and later went on to cover jQuery's support for custom events, which offer a similar solution for implementing Publish/Subscribe. This same pattern can be used when writing jQuery plug-ins.

The basic idea here is that objects in a page can publish event notifications when something interesting occurs in our application. Other objects then subscribe to (or listen) for these events and respond accordingly. This results in the logic for our application being significantly more decoupled, as each object no longer needs to directly communicate with another.

In the following jQuery UI Widget Factory pattern, we'll implement a basic custom event-based Publish/Subscribe system that allows our plug-in to subscribe to event notifications from the rest of our application, which will be responsible for publishing them.

```
/*!
 * jQuery custom-events plugin boilerplate
 * Author: DevPatch
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

// In this pattern, we use jQuery's custom events to add
// pub/sub (publish/subscribe) capabilities to widgets.
// Each widget would publish certain events and subscribe
// to others. This approach effectively helps to decouple
// the widgets and enables them to function independently.

;(function ( $, window, document, undefined ) {
    $.widget( "ao.eventStatus", {
        options: {

        },

        _create : function() {
            var self = this;

            //self.element.addClass( "my-widget" );

            //subscribe to "myEventStart"
            self.element.on( "myEventStart", function( e ) {
                console.log( "event start" );
            });

            //subscribe to "myEventEnd"
            self.element.on( "myEventEnd", function( e ) {
                console.log( "event end" );
            });

            //unsubscribe to "myEventStart"
            //self.element.off( "myEventStart", function(e){
                ///console.log( "unsubscribed to this event" );
            //});
        },

        destroy: function(){
            $.Widget.prototype.destroy.apply( this, arguments );
        },
    });
})( jQuery, window , document );

// Publishing event notifications
// $( ".my-widget" ).trigger( "myEventStart");
// $( ".my-widget" ).trigger( "myEventEnd" );
```

Usage:

```
var el = $( "#elem" );
el.eventStatus();
el.eventStatus().trigger( "myEventStart" );
```

Further Reading

- Communication Between jQuery UI Widgets, Benjamin Sternthal

Prototypal Inheritance with the DOM-to-Object Bridge Pattern

As covered earlier, in JavaScript, we don't have the traditional notion of classes that we would find in other classical programming languages, but we do have prototypal inheritance. With prototypal inheritance, an object inherits from another object. We can apply this concept to jQuery plug-in development.

Yepnope.js author Alex Sexton and jQuery team member Scott Gonzalez have looked at this topic in detail. In sum, they discovered that for organized modular development, clearly separating the object that defines the logic for a plug-in from the plug-in generation process itself can be beneficial.

The benefit is that testing our plug-in's code becomes significantly easier, and we are also able to adjust the way things work behind the scenes without altering the way that any object APIs we implement are used.

In Sexton's article on this topic, he implemented a bridge that enables us to attach our general logic to a particular plug-in, which we've implemented in the pattern below.

One of the other advantages of this pattern is that we don't have to constantly repeat the same plug-in initialization code, thus ensuring that the concepts behind DRY development are maintained. Some developers might also find this pattern easier to read than others.

```
/*!
 * jQuery prototypal inheritance plugin boilerplate
 * Author: Alex Sexton, Scott Gonzalez
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */


// myObject - an object representing a concept we wish to model
// (e.g. a car)
var myObject = {
  init: function( options, elem ) {
    // Mix in the passed-in options with the default options
    this.options = $.extend( {}, this.options, options );

    // Save the element reference, both as a jQuery
    // reference and a normal reference
    this.elem  = elem;
    this.$elem = $( elem );

    // Build the DOM's initial structure
    this._build();

    // return this so that we can chain and use the bridge with less code.
    return this;
  },
  options: {
    name: "No name"
  },
  _build: function(){
    //this.$elem.html( "<h1>"+this.options.name+"</h1>" );
  },
  myMethod: function( msg ){
    // We have direct access to the associated and cached
    // jQuery element
    // this.$elem.append( "<p>"+msg+"</p>" );
  }
};


// Object.create support test, and fallback for browsers without it
if ( typeof Object.create !== "function" ) {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}


// Create a plug-in based on a defined object
$.plugin = function( name, object ) {
  $.fn[name] = function( options ) {
    return this.each(function() {
      if ( ! $.data( this, name ) ) {
        $.data( this, name, Object.create( object ).init(
        options, this ) );
      }
    });
  };
};
```

Usage:

```
  $.plugin( "myobj", myObject );

  $("#elem").myobj( {name: "John"} );

  var collection = $( "#elem" ).data( "myobj" );
  collection.myMethod( "I am a method");
```

Further Reading

- Using Inheritance Patterns To Organize Large jQuery Applications, Alex Sexton
- How to Manage Large Applications With jQuery or Whatever (further discussion), Alex Sexton
- Practical Example of the Need for Prototypal Inheritance, Neeraj Singh
- Prototypal Inheritance in JavaScript, Douglas Crockford

jQuery UI Widget Factory Bridge Pattern

If you liked the idea of generating plug-ins based on objects in the last design pattern, then you might be interested in a method found in the jQuery UI Widget Factory called `$.widget.bridge`.

This bridge basically serves as a middle layer between a JavaScript object that is created using `$.widget` and the jQuery core API, providing a more built-in solution to achieving object-based plug-in definition. Effectively, we're able to create stateful plug-ins using a custom constructor.

Moreover, `$.widget.bridge` provides access to a number of other capabilities, including the following:

- Both public and private methods are handled as one would expect in classical OOP (i.e., public methods are exposed, while calls to private methods are not possible).
- Automatic protection against multiple initializations.
- Automatic generation of instances of a passed object and storage of them within the selection's internal `$.data` cache.
- Options can be altered post-initialization.

For further information on how to use this pattern, please see the inline comments below:

```
/*!
 * jQuery UI Widget factory "bridge" plugin boilerplate
 * Author: @erichynds
 * Further changes, additional comments: @addyosmani
 * Licensed under the MIT license
 */


// a "widgetName" object constructor
// required: this must accept two arguments,
// options: an object of configuration options
// element: the DOM element the instance was created on
var widgetName = function( options, element ){
  this.name = "myWidgetName";
  this.options = options;
  this.element = element;
  this._init();
}

// the "widgetName" prototype
widgetName.prototype = {

    // _create will automatically run the first time this
    // widget is called
    _create: function(){
        // creation code
    },

    // required: initialization logic for the plug-in goes into _init
    // This fires when our instance is first created and when
    // attempting to initialize the widget again (by the bridge)
    // after it has already been initialized.
    _init: function(){
        // init code
    },

    // required: objects to be used with the bridge must contain an
    // "option". Post-initialization, the logic for changing options
    // goes here.
    option: function( key, value ){

        // optional: get/change options post initialization
        // ignore if you don't require them.

        // signature: $("#foo").bar({ cool:false });
        if( $.isPlainObject( key ) ){
            this.options = $.extend( true, this.options, key );

        // signature: $( "#foo" ).option( "cool" ); - getter
        } else if ( key && typeof value === "undefined" ){
            return this.options[ key ];
        // signature: $( "#foo" ).bar("option", "baz", false );
        } else {
            this.options[ key ] = value;
        }

        // required: option must return the current instance.
        // When re-initializing an instance on elements, option
        // is called first and is then chained to the _init method.
        return this;
    },

    // notice no underscore is used for public methods
    publicFunction: function(){
        console.log( "public function" );
    },

    // underscores are used for private methods
    _privateFunction: function(){
        console.log( "private function" );
    }
};
```

Usage:

```
    // connect the widget obj to jQuery's API under the "foo" namespace
    $.widget.bridge( "foo", widgetName );

    // create an instance of the widget for use
    var instance = $( "#foo" ).foo({
       baz: true
    });

    // our widget instance exists in the elem's data
    // Outputs: #elem
    console.log(instance.data( "foo" ).element);

    // bridge allows us to call public methods
    // Outputs: "public method"
    instance.foo("publicFunction");

    // bridge prevents calls to internal methods
    instance.foo("_privateFunction");
```

Further Reading

- Using $.widget.bridge Outside of the Widget Factory, Eric Hynds

jQuery Mobile Widgets with the Widget Factory

jQuery Mobile is a jQuery project framework that encourages the design of ubiquitous web applications that work both on popular mobile devices and platforms and on the desktop. Rather than writing unique applications for each device or OS, we simply write the code once, and it should ideally run on many of the A-, B- and C-grade browsers out there at the moment.

The fundamentals behind jQuery Mobile can also be applied to plug-in and widget development.

What's interesting in this next pattern is that although there are small, subtle differences in writing a "mobile"-optimized widget, those familiar with using the jQuery UI Widget Factory pattern from earlier should be able to grasp this in next to no time.

The mobile-optimized widget below has a number of interesting differences than the standard UI widget pattern we saw earlier:

- `$.mobile.widget` is referenced as an existing widget prototype from which to inherit. For standard widgets, passing through any such prototype is unnecessary for basic development, but using this jQuery-mobile specific widget prototype provides internal access to further "options" formatting.
- In `_create()`, a guide is provided on how the official jQuery Mobile widgets handle element selection, opting for a role-based approach that better fits the jQM markup. This isn't at all to say that standard selection isn't recommended, only that this approach might make more sense given the structure of jQuery Mobile pages.
- Guidelines are also provided in comment form for applying our plug-in methods on `pagecreate` as well as for selecting the plug-in application via data roles and data attributes.

```
/*!
 * (jQuery mobile) jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)
 * Author: @scottjehl
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

;(function ( $, window, document, undefined ) {

    // define a widget under a namespace of our choice
    // here "mobile" has been used in the first argument
    $.widget( "mobile.widgetName", $.mobile.widget, {

        // Options to be used as defaults
        options: {
            foo: true,
            bar: false
        },

        _create: function() {
            // _create will automatically run the first time this
            // widget is called. Put the initial widget set-up code
            // here, then we can access the element on which
            // the widget was called via this.element
            // The options defined above can be accessed via
            // this.options
            // var m = this.element,
            // p = m.parents( ":jqmData(role="page")" ),
            // c = p.find( ":jqmData(role="content")" )
        },

        // Private methods/props start with underscores
        _dosomething: function(){ ... },

        // Public methods like these below can can be called
        // externally:
        // $("#myelem").foo( "enable", arguments );

        enable: function() { ... },

        // Destroy an instantiated plug-in and clean up modifications
        // the widget has made to the DOM
        destroy: function () {
            // this.element.removeStuff();
            // For UI 1.8, destroy must be invoked from the
            // base widget
            $.Widget.prototype.destroy.call( this );
            // For UI 1.9, define _destroy instead and don't
            // worry about calling the base widget
        },

        methodB: function ( event ) {
            //_trigger dispatches callbacks the plug-in user can
            // subscribe to
            // signature: _trigger( "callbackName" , [eventObject],
            //  [uiObject] )
            // e.g. this._trigger( "hover", e /*where e.type ==
            // "mouseenter"*/, { hovered: $(e.target)});
            this._trigger( "methodA", event, {
                key: value
            });
        },

        methodA: function ( event ) {
            this._trigger( "dataChanged", event, {
                key: value
            });
        },

        // Respond to any changes the user makes to the option method
        _setOption: function ( key, value ) {
            switch ( key ) {
            case "someValue":
                // this.options.someValue = doSomethingWith( value );
                break;
            default:
                // this.options[ key ] = value;
                break;
            }
```

```
                // For UI 1.8, _setOption must be manually invoked from
                // the base widget
                $.Widget.prototype._setOption.apply(this, arguments);
                // For UI 1.9 the _super method can be used instead
                // this._super( "_setOption", key, value );
            }
        });

    })( jQuery, window, document );
```

Usage:

```
    var instance = $( "#foo" ).widgetName({
      foo: false
    });

    instance.widgetName( "methodB" );
```

We can also self-initialize this widget whenever a new page in jQuery Mobile is created. jQuery Mobile's *page* plug-in dispatches a `create` event when a jQuery Mobile page (found via the `data-role="page"` attribute) is first initialized. We can listen for that event (called `pagecreate`) and run our plug-in automatically whenever a new page is created.

```
    $(document).on("pagecreate", function ( e ) {
        // In here, e.target refers to the page that was created
        // (it's the target of the pagecreate event)
        // So, we can simply find elements on this page that match a
        // selector of our choosing, and call our plug-in on them.
        // Here's how we'd call our "foo" plug-in on any element with a
        // data-role attribute of "foo":
        $(e.target).find( "[data-role="foo"]" ).foo( options );

        // Or, better yet, let's write the selector accounting for the configurable
        // data-attribute namespace
        $( e.target ).find( ":jqmData(role="foo")" ).foo( options );
    });
```

We can now simply reference the script containing our widget and `pagecreate` binding in a page running the jQuery Mobile site, and it will automatically run like any other jQuery Mobile plug-in.

RequireJS and the jQuery UI Widget Factory

As we covered in Chapter 11, RequireJS is an AMD-compatible script loader that provides a clean solution for encapsulating application logic inside manageable modules.

It's able to load modules in the correct order (through its order plug-in), simplifies the process of combining scripts via its excellent r.js optimizer, and provides the means for defining dynamic dependencies on a per-module basis.

In the boilerplate pattern below, we demonstrate how an AMD (and thus RequireJS) compatible jQuery UI widget can be defined that does the following:

- Allows the definition of widget module dependencies, building on top of the previous jQuery UI Widget Factory pattern presented earlier.
- Demonstrates one approach to passing in HTML template assets for creating templated widgets (using Underscore.js microtemplating).
- Includes a quick tip on adjustments that we can make to our widget module if we wish to later pass it through to the RequireJS optimizer.

```
/*!
 * jQuery UI Widget + RequireJS module boilerplate (for 1.8/9+)
 * Authors: @jrburke, @addyosmani
 * Licensed under the MIT license
 */

// Note from James:
//
// This assumes we are using the RequireJS+jQuery file, and
// that the following files are all in the same directory:
//
// - require-jquery.js
// - jquery-ui.custom.min.js (custom jQuery UI build with widget factory)
// - templates/
//    - asset.html
// - ao.myWidget.js

// Then we can construct the widget as follows:

// ao.myWidget.js file:
define( "ao.myWidget", ["jquery", "text!templates/asset.html", "underscore",
"jquery-ui.custom.min"], function ( $, assetHtml, _ ) {

    // define our widget under a namespace of our choice
    // "ao" is used here as a demonstration
    $.widget( "ao.myWidget", {

        // Options to be used as defaults
        options: {},

        // Set up widget (e.g. create element, apply theming,
        // bind events, etc.)
        _create: function () {

            // _create will automatically run the first time
            // this widget is called. Put the initial widget
            // set-up code here, then we can access the element
            // on which the widget was called via this.element.
            // The options defined above can be accessed via
            // this.options

            // this.element.addStuff();
            // this.element.addStuff();

            // We can then use Underscore templating with
            // with the assetHtml that has been pulled in
            // var template = _.template( assetHtml );
            // this.content.append( template({}) );
        },

        // Destroy an instantiated plug-in and clean up modifications
        // that the widget has made to the DOM
        destroy: function () {
            // this.element.removeStuff();
            // For UI 1.8, destroy must be invoked from the base
            // widget
            $.Widget.prototype.destroy.call( this );
            // For UI 1.9, define _destroy instead and don't worry
            // about calling the base widget
        },

        methodB: function ( event ) {
            // _trigger dispatches callbacks the plug-in user can
            // subscribe to
            // signature: _trigger( "callbackName" , [eventObject],
            // [uiObject] )
            this._trigger( "methodA", event, {
                key: value
            });
        },

        methodA: function ( event ) {
            this._trigger("dataChanged", event, {
                key: value
            });
        },

        // Respond to any changes the user makes to the option method
        _setOption: function ( key, value ) {
            switch (key) {
            case "someValue":
                // this.options.someValue = doSomethingWith( value );
                break;
            default:
                // this.options[ key ] = value;
                break;
            }

            // For UI 1.8, _setOption must be manually invoked from
            // the base widget
            $.Widget.prototype._setOption.apply( this, arguments );
            // For UI 1.9 the _super method can be used instead
            // this._super( "_setOption", key, value );
        }

    });
});
```

Usage

Here is the index.html code:

```
<script data-main="scripts/main" src="requirejs">
</script>
```

Here is the main.js code:

```
require({

    paths: {
        "jquery": "https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min",
        "jqueryui": "https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.18/jquery-ui.min",
        "boilerplate": "../patterns/jquery.widget-factory.requirejs.boilerplate"
    }
}, ["require", "jquery", "jqueryui", "boilerplate"],
function (req, $) {

    $(function () {

        var instance = $("#elem").myWidget();
        instance.myWidget("methodB");

    });
});
```

Further Reading

- Using RequireJS with jQuery, Rebecca Murphey
- Fast Modular Code With jQuery and RequireJS, James Burke
- jQuery's Best Friends, Alex Sexton
- Managing Dependencies With RequireJS, Ruslan Matveev

Globally and Per-Call Overridable Options (Best Options Pattern)

For our next pattern, we'll look at an optimal approach to configuring options and defaults for a plug-in. The way most of us are probably familiar with defining plug-in options is to pass through an object literal of defaults to `$.extend()`, as demonstrated in our basic plug-in boilerplate.

If, however, we're working with a plug-in with many customizable options that we would like users to be able to override either globally or on a per-call level, then we can structure things a little more optimally.

Instead, by referring to an options object defined within the plug-in namespace explicitly (for example, `$fn.pluginName.options`) and merging this with any options passed through to the plug-in when it is initially invoked, users have the option of either passing options through during plug-in initialization or overriding options outside of the plug-in (as demonstrated here).

```
/*!
 * jQuery "best options" plugin boilerplate
 * Author: @cowboy
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */


;(function ( $, window, document, undefined ) {

    $.fn.pluginName = function ( options ) {

        // Here's a best practice for overriding "defaults"
        // with specified options. Note how, rather than a
        // regular defaults object being passed as the second
        // parameter, we instead refer to $.fn.pluginName.options
        // explicitly, merging it with the options passed directly
        // to the plug-in. This allows us to override options both
        // globally and on a per-call level.

        options = $.extend( {}, $.fn.pluginName.options, options );

        return this.each(function () {

            var elem = $(this);

        });
    };

    // Globally overriding options
    // Here are our publicly accessible default plug-in options
    // that are available in case the user doesn't pass in all
    // of the values expected. The user is given a default
    // experience but can also override the values as necessary.
    // e.g. $fn.pluginName.key ="otherval";

    $.fn.pluginName.options = {

        key: "value",
        myMethod: function ( elem, param ) {

        }
    };

})( jQuery, window, document );
```

Usage:

```
$("#elem").pluginName({
  key: "foobar"
});
```

Further Reading

- jQuery Pluginization and the accompanying gist written by Ben Alman

A Highly Configurable and Mutable Plug-in Pattern

In this pattern, similar to Alex Sexton's prototypal inheritance plug-in pattern, logic for our plug-in isn't nested in a jQuery plug-in itself. We instead define our plug-in logic using a constructor and an object literal defined on its prototype. jQuery is then used for the actual instantiation of the plug-in object.

Customization is taken to the next level by employing two little tricks, one of which we've seen in previous patterns:

- Options can be overridden both globally and per collection of elements.
- Options can be customized on a *per-element* level through HTML5 data attributes (as shown below). This facilitates plug-in behavior that can be applied to a collection of elements but then customized inline without the need to instantiate each element with a different default value.

We don't see the latter option in the wild too often, but it can be a significantly cleaner solution (as long as we don't mind the inline approach). If you're wondering where this could be useful, imagine writing a draggable plug-in for a large set of elements. We could go about customizing their options as follows:

```
$( ".item-a" ).draggable( {"defaultPosition":"top-left"} );
$( ".item-b" ).draggable( {"defaultPosition":"bottom-right"} );
$( ".item-c" ).draggable( {"defaultPosition":"bottom-left"} );
//etc
```

But using our patterns inline approach, the following would be possible:

```
$( ".items" ).draggable();
```

```
html
<li class="item" data-plugin-options="{"defaultPosition":"top-left"}"></div>
<li class="item" data-plugin-options="{"defaultPosition":"bottom-left"}"></div>
```

And so on. We may well have a preference for one of these approaches, but it is just another variation worth being aware of.

```
/*
 * "Highly configurable" mutable plugin boilerplate
 * Author: @markdalgleish
 * Further changes, comments: @addyosmani
 * Licensed under the MIT license
 */


// Note that with this pattern, as per Alex Sexton's, the plug-in logic
// hasn't been nested in a jQuery plug-in. Instead, we just use
// jQuery for its instantiation.
```

```
;(function( $, window, document, undefined ){

  // our plug-in constructor
  var Plugin = function( elem, options ){
      this.elem = elem;
      this.$elem = $(elem);
      this.options = options;

      // This next line takes advantage of HTML5 data attributes
      // to support customization of the plug-in on a per-element
      // basis. For example,
      // <div class=item" data-plugin-options="{"message":"Goodbye World!"}"></div>
      this.metadata = this.$elem.data( "plugin-options" );
    };

  // the plug-in prototype
  Plugin.prototype = {
    defaults: {
      message: "Hello world!"
    },

    init: function() {
      // Introduce defaults that can be extended either
      // globally or using an object literal.
      this.config = $.extend( {}, this.defaults, this.options,
      this.metadata );

      // Sample usage:
      // Set the message per instance:
      // $( "#elem" ).plugin( { message: "Goodbye World!"} );
      // or
      // var p = new Plugin( document.getElementById( "elem" ),
      // { message: "Goodbye World!"}).init()
      // or, set the global default message:
      // Plugin.defaults.message = "Goodbye World!"

      this.sampleMethod();
      return this;
    },

    sampleMethod: function() {
      // e.g. show the currently configured message
      // console.log(this.config.message);
    }
  }

  Plugin.defaults = Plugin.prototype.defaults;

  $.fn.plugin = function( options ) {
    return this.each(function() {
      new Plugin( this, options ).init();
    });
  };
```

```
        // optional: window.Plugin = Plugin;

    })( jQuery, window , document );
```

Usage:

```
    $("#elem").plugin({
      message: "foobar"
    });
```

Further Reading

- Creating Highly Configurable jQuery Plug-ins, Mark Dalgleish
- Writing Highly Configurable jQuery Plug-ins, Part 2, Mark Dalgleish

What Makes a Good Plug-in Beyond Patterns?

At the end of the day, design patterns are just one facet to writing maintainable jQuery plug-ins. There are a number of other factors worth considering, and I would like to share my own criteria for selecting third-party plug-ins to address some of the other concerns. I hope this helps increase the overall quality of your plug-in projects:

Quality

Adhere to best practices with respect to both the JavaScript and jQuery that you write. Are efforts being made to lint the plug-in code using either jsHint or jsLint? Is the plug-in written optimally?

Code Style

Does the plug-in follow a consistent code style guide such as the jQuery Core Style Guidelines? If not, is your code at least relatively clean and readable?

Compatibility

Which versions of jQuery is the plug-in compatible with? Has it been tested with the latest jQuery-git builds or latest stable? If the plug-in was written before jQuery 1.6, then it might have issues with attributes and properties, because the way they were approached changed in that release.

New versions of jQuery offer improvements and opportunities for the jQuery project to improve on what the core library offers. With this comes occasional breakages (mainly in major releases) as we move toward a better way of doing things. I'd like to see plug-in authors update their code when necessary or, at a minimum, test their plug-ins with new versions to make sure everything works as expected.

Reliability

The plug-in should come with its own set of unit tests. Not only do these prove it actually functions as expected, but they can also improve the design without breaking it for end users. I consider unit tests essential for any serious jQuery plug-in that is meant for a production environment, and they're not that hard to write.

For an excellent guide to automated JavaScript testing with QUnit, you may be interested in "Automating JavaScript Testing With QUnit", by Jörn Zaefferer.

Performance

If the plug-in needs to perform tasks that require extensive processing or heavy manipulation of the DOM, one should follow best practices for benchmarking to help minimize this. Use jsPerf.com to test segments of the code to determine how well it performs in different browsers and discover what, if anything, might be optimized further.

Documentation

If the intention is for other developers to use the plug-in, ensure that it's well documented. Document the API and how the plug-in is to be used. What methods and options does the plug-in support? Does it have any gotchas that users need to be aware of? If users cannot figure out how to use the plug-in, they'll likely look for an alternative. It is also of great help to comment your plug-in code. This is by far the best gift you can offer other developers. If someone feels they can navigate your code base well enough to use it or improve it, then you've done a good job.

Likelihood of maintenance

When releasing a plug-in, estimate how much time may be required for maintenance and support. We all love to share our plug-ins with the community, but one needs to set expectations for one's ability to answer questions, address issues, and make continuous improvements. This can be done simply by stating the project intentions for maintenance support upfront in the *README* file.

Conclusions

In this chapter so far, we have explored several time-saving design patterns and best practices that can be employed to improve how jQuery plug-ins can be written. Some are better suited to certain use cases than others, but I hope on the whole these patterns are useful.

Remember, when selecting a pattern, it is important to be practical. Don't use a plug-in pattern just for the sake of it, rather, invest time in understanding the underlying structure, and establish how well it solves your problem or fits the component you're trying to build.

Namespacing Patterns

In this section, we're going to explore patterns for namespacing in JavaScript. Namespaces can be considered a logical grouping of units of code under a unique identifier. The identifier can be referenced in many namespaces, and each identifier can itself contain a hierarchy of its own nested (or sub) namespaces.

In application development, we employ namespaces for a number of important reasons. In JavaScript, they help us avoid *collisions* with other objects or variables in the global namespace. They're also extremely useful for helping organize blocks of functionality in a code base so that it can be more easily referenced and used.

Namespacing any serious script or application is critical, as it's important to safeguard our code from breaking in the event of another script on the page using the *same* variable or method names we are. With the number of *third-party* tags regularly injected into pages these days, this can be a common problem we all need to tackle at some point in our careers. As a well-behaved "citizen" of the global namespace, it's also imperative that we try our best not to prevent other developers' scripts from executing due to the same issues.

While JavaScript doesn't really have built-in support for namespaces like other languages, it does have objects and closures which can be used to achieve a similar effect.

Namespacing Fundamentals

Namespaces can be found in almost any serious JavaScript application. Unless we're working with a simple code snippet, it's imperative that we do our best to ensure that we're implementing namespacing correctly, as it's not just simple to pick up, it'll also avoid third-party code clobbering our own. The patterns we'll be examining in this section are:

1. Single global variables
2. Prefix namespacing
3. Object literal notation
4. Nested namespacing
5. Immediately-invoked Function
6. Expressions
7. Namespace injection

Single Global Variables

One popular pattern for namespacing in JavaScript is opting for a single global variable as our primary object of reference. A skeleton implementation of this where we return an object with functions and properties can be found below:

```
var myApplication =  (function () {
        function(){
            //...
        },
        return{
            //...
        }
})();
```

Although this works for certain situations, the biggest challenge with the single global variable pattern is ensuring that no one else has used the same global variable name as we have in the page.

Prefix Namespacing

One solution to the above problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is we select a unique prefix namespace we wish to use (in this example, `myApplication_`) and then define any methods, variables, or other objects after the prefix as follows:

```
var myApplication_propertyA = {};
var myApplication_propertyB = {};
function myApplication_myMethod(){
  //...
}
```

This is effective from the perspective of decreasing the chances of a particular variable existing in the global scope, but remember that a uniquely named object can have the same effect.

This aside, the biggest issue with the pattern is that it can result in a large number of global objects once our application starts to grow. There is also quite a heavy reliance on our prefix not being used by any other developers in the global namespace, so be careful if opting to use this.

For more on Peter's views about the single global variable pattern, read his excellent post on them: *http://michaux.ca/articles/javascript-namespacing*.

Object Literal Notation

Object literal notation, which we also cover in the Module pattern section, can be thought of as an object containing a collection of key-value pairs with a colon separating each pair of keys and values, where keys can also represent new namespaces.

```
var myApplication = {

    // As we've seen, we can easily define functionality for
    // this object literal..
    getInfo:function(){
      //...
    },

    // but we can also populate it to support
    // further object namespaces containing anything
    // anything we wish:
    models : {},
    views : {
        pages : {}
    },
    collections : {}
};
```

One can also opt for adding properties directly to the namespace:

```
myApplication.foo = function(){
    return "bar";
}

myApplication.utils = {
    toString:function(){
        //...
    },
    export: function(){
        //...
    }
}
```

Object literals have the advantage of not polluting the global namespace but assist in organizing code and parameters logically. They are truly beneficial if you wish to create easily readable structures that can be expanded to support deep nesting. Unlike simple global variables, object literals often also take into account tests for the existence of a variable by the same name, so the chances of collision occurring are significantly reduced.

The next sample demonstrates a number of different ways in which you can check to see if an object namespace already exists, defining it if it doesn't.

```
// This doesn't check for existence of "myApplication" in
// the global namespace. Bad practice as we can easily
// clobber an existing variable/namespace with the same name
var myApplication = {};

// The following options *do* check for variable/namespace existence.
// If already defined, we use that instance, otherwise we assign a new
// object literal to myApplication.
//
// Option 1: var myApplication = myApplication || {};
// Option 2  if( !MyApplication ){ MyApplication = {} };
// Option 3: window.myApplication || ( window.myApplication = {} );
// Option 4: var myApplication = $.fn.myApplication = function() {};
// Option 5: var myApplication = myApplication === undefined ? {} : myApplication;
```

You'll often see developers opting for Option 1 or Option 2—they are both straightforward and are equivalent in terms of their end result.

Option 3 assumes that you're working in the global namespace, but it can also be written as:

```
myApplication || (myApplication = {});
```

This variation assumes that `myApplication` has already been initialized, so it's only really useful for a parameter/argument scenario, as in the following example:

```
function foo() {
  myApplication || ( myApplication = {} );
}

// myApplication hasn't been initialized,
// so foo() throws a ReferenceError

foo();

// However accepting myApplication as an
// argument

function foo( myApplication ) {
  myApplication || ( myApplication = {} );
}

foo();

// Even if myApplication === undefined, there is no error
// and myApplication gets set to {} correctly
```

Options 4 can be useful for writing jQuery plug-ins where:

```
// If we were to define a new plug-in..
var myPlugin = $.fn.myPlugin = function() { ... };

// Then later rather than having to type:
$.fn.myPlugin.defaults = {};

// We can do:
myPlugin.defaults = {};
```

This results in better compression (minification) and can save on scope lookups.

Option 5 is a little similar to Option 4, but is a long form that evaluates whether `myApplication` is `undefined` inline—such that it's defined as an object if not—and set to an existing value for `myApplication` if so.

It is shown just for the sake of being thorough, but in most situations, Options 1–4 will more than suffice for most needs.

There is, of course, a great deal of variance in how and where object literals are used for organizing and structuring code. For smaller applications wishing to expose a nested API for a particular self-enclosed module, you may just find yourself using the Revealing Module pattern, which we covered earlier in the book:

```
var namespace = (function () {

    // defined within the local scope
    var privateMethod1 = function () { /* ... */ },
        privateMethod2 = function () { /* ... */ }
        privateProperty1 = "foobar";

    return {

        // the object literal returned here can have as many
        // nested depths as we wish, however as mentioned,
        // this way of doing things works best for smaller,
        // limited-scope applications in my personal opinion
        publicMethod1: privateMethod1,

        // nested namespace with public properties
        properties:{
            publicProperty1: privateProperty1
        },

        // another tested namespace
        utils:{
            publicMethod2: privateMethod2
        }
        ...
    }
})();
```

The benefit of using object literals here is that they offer us a very elegant key/value syntax to work with; one where we're able to easily encapsulate any distinct logic or functionality for our application in a way that clearly separates it from others and provides a solid foundation for extending our code.

```
var myConfig = {

    language: "english",

    defaults: {
        enableGeolocation: true,
        enableSharing: false,
        maxPhotos: 20
    },

    theme: {
        skin: "a",
        toolbars: {
            index: "ui-navigation-toolbar",
            pages: "ui-custom-toolbar"
        }
        }

    }
```

Note that JSON is a subset of object literal notation, and there are really only minor syntactical differences between it and the above (e.g., JSON keys must be strings). If, for any reason, one wishes to use JSON for storing configuration data instead (e.g., for simpler storage when sending to the backend), feel free to. For more on the object literal pattern, I recommend reading Rebecca Murphey's excellent article on the topic, as she covers a few areas we didn't touch upon.

Nested Namespacing

An extension of the object literal pattern is nested namespacing. It's another common pattern used that offers a lower risk of collision due to the fact that even if a namespace already exists, it's unlikely the same nested children do.

Does this look familiar?

```
YAHOO.util.Dom.getElementsByClassName("test");
```

Older versions of Yahoo!'s YUI library use the nested object namespacing pattern regularly. During my time as an engineer at AOL, we also used this pattern in many of our larger applications. A sample implementation of nested namespacing may look like this:

```
var myApp =  myApp || {};

// perform a similar existence check when defining nested
// children
myApp.routers = myApp.routers || {};
myApp.model = myApp.model || {};
myApp.model.special = myApp.model.special || {};

// nested namespaces can be as complex as required:
// myApp.utilities.charting.html5.plotGraph(/*..*/);
// myApp.modules.financePlanner.getSummary();
// myApp.services.social.facebook.realtimeStream.getLatest();
```

The above differs from how YUI3 approaches namespacing as modules there use a sandboxed API host object with far less and far shallower namespacing.

We can also opt to declare new nested namespaces/properties as indexed properties as follows:

```
myApp["routers"] = myApp["routers"] || {};
myApp["models"] = myApp["models"] || {};
myApp["controllers"] = myApp["controllers"] || {};
```

Both options are readable, organized, and offer a relatively safe way of namespacing our application in a similar fashion to what we may be used to in other languages. The only real caveat, however, is that it requires our browser's JavaScript engine first locating the `myApp` object and then digging down until it gets to the function we actually wish to use.

This can mean an increased amount of work to perform lookups; however, developers such as Juriy Zaytsev have previously tested and found the performance differences between single object namespacing versus the "nested" approach to be quite negligible.

Immediately Invoked Function Expressions (IIFE)s

Earlier in the book, we briefly covered the concept of an immediately invoked function expression; IIFE, which is effectively an unnamed function, immediately invoked after it's been defined. If it sounds familiar it's because you may have previous come across it referred to as a self-executing (or self-invoked) anonymous function, however I personally feel Ben Alman's IIFE naming is more accurate. In JavaScript, because both variables and functions explicitly defined within such a context may only be accessed inside of it, function invocation provides an easy means to achieving privacy.

IIFEs are a popular approach to encapsulating application logic to protect it from the global namespace but also have their use in the world of namespacing.

Examples of IIFEs can be found below:

```
// an (anonymous) immediately-invoked function expression
(function () { /*...*/})();

// a named immediately-invoked function expression
(function foobar () { /*..*/}());

// this is technically a self-executing function which is quite different
function foobar () { foobar(); }
```

while a slightly more expanded version of the first example might look like:

```
var namespace = namespace || {};

// here a namespace object is passed as a function
// parameter, where we assign public methods and
// properties to it
(function( o ){
    o.foo = "foo";
    o.bar = function(){
        return "bar";
    };
})( namespace );

console.log( namespace );
```

Whilst readable, this example could be significantly expanded on to address common development concerns such as defined levels of privacy (public/private functions and variables) as well as convenient namespace extension. Let's go through some more code:

```
// namespace (our namespace name) and undefined are passed here
// to ensure 1. namespace can be modified locally and isn't
// overwritten outside of our function context
// 2. the value of undefined is guaranteed as being truly
// undefined. This is to avoid issues with undefined being
// mutable pre-ES5.

;(function ( namespace, undefined ) {

    // private properties
    var foo = "foo",
        bar = "bar";

    // public methods and properties
    namespace.foobar = "foobar";
    namespace.sayHello = function () {
        speak( "hello world" );
    };

    // private method
    function speak(msg) {
        console.log( "You said: " + msg );
    };

    // check to evaluate whether "namespace" exists in the
    // global namespace - if not, assign window.namespace an
    // object literal

}( window.namespace = window.namespace || {} ));


// we can then test our properties and methods as follows

// public

// Outputs: foobar
console.log( namespace.foobar );

// Outputs: hello world
namescpace.sayHello();

// assigning new properties
namespace.foobar2 = "foobar";

// Outputs: foobar
console.log( namespace.foobar2 );
```

Extensibility is of course key to any scalable namespacing pattern and IIFEs can be used to achieve this quite easily. In the below example, our "namespace" is once again passed as an argument to our anonymous function and is then extended (or decorated) with further functionality:

```
// let's extend the namespace with new functionality
(function( namespace, undefined ){

    // public method
    namespace.sayGoodbye = function () {
        console.log( namespace.foo );
        console.log( namespace.bar );
        speak( "goodbye" );
    }
}( window.namespace = window.namespace || {});

// Outputs: goodbye
namespace.sayGoodbye();
```

If you would like to find out more about this pattern, I recommend reading Ben's IIFE post for more information.

Namespace Injection

Namespace injection is another variation on the IIFE in which we "inject" the methods and properties for a specific namespace from within a function wrapper using `this` as a namespace proxy. The benefit this pattern offers is easy application of functional behavior to multiple objects or namespaces and can come in useful when applying a set of base methods to be built on later (e.g., getters and setters).

The disadvantages of this pattern are that there may be easier or more optimal approaches to achieving this goal (e.g., deep object extension or merging), which I cover earlier in the section.

Below we can see an example of this pattern in action, where we use it to populate the behavior for two namespaces: one initially defined (`utils`) and another which we dynamically create as a part of the functionality assignment for `utils` (a new namespace called `tools`).

```
var myApp = myApp || {};
myApp.utils =  {};

(function () {
  var val = 5;

  this.getValue = function () {
      return val;
  };

  this.setValue = function( newVal ) {
      val = newVal;
  }

  // also introduce a new sub-namespace
  this.tools = {};

}).apply( myApp.utils );
```

```
// inject new behaviour into the tools namespace
// which we defined via the utilities module

(function () {
    this.diagnose = function(){
        return "diagnosis";
    }
}).apply( myApp.utils.tools );

// note, this same approach to extension could be applied
// to a regular IIFE, by just passing in the context as
// an argument and modifying the context rather than just
// "this"

// Usage:

// Outputs our populated namespace
console.log( myApp );

// Outputs: 5
console.log( myApp.utils.getValue() );

// Sets the value of `val` and returns it
myApp.utils.setValue( 25 );
console.log( myApp.utils.getValue() );

// Testing another level down
console.log( myApp.utils.tools.diagnose() );
```

Angus Croll has also suggested the idea of using the call API to provide a natural separation between contexts and arguments previously. This pattern can feel a lot more like a module creator, but as modules still offer an encapsulation solution, we'll briefly cover it for the sake of thoroughness:

```
// define a namespace we can use later
var ns = ns || {},
    ns2 = ns2 || {};

// the module/namespace creator
var creator = function( val ){

    var val = val || 0;

    this.next = function () {
        return val++
    };

    this.reset = function () {
        val = 0;
    }
}

creator.call( ns );

// ns.next, ns.reset now exist
```

```
creator.call( ns2 , 5000 );

// ns2 contains the same methods
// but has an overridden value for val
// of 5000
```

As mentioned, this type of pattern is useful for assigning a similar base set of functionality to multiple modules or namespaces. I would however only really suggest using it where explicitly declaring functionality within an object/closure for direct access doesn't make sense.

Advanced Namespacing Patterns

We'll now explore some advanced patterns and utilities that I have found invaluable when working on larger applications, some of which have required a rethink of traditional approaches to application namespacing. I'll note that I am not advocating any of the following as *the* way to namespace, but rather ways that I have found work in practice.

Automating Nested Namespacing

As we've reviewed, nested namespaces can provide an organized hierarchy of structure for a unit of code. An example of such a namespace could be the following: ***application.utilities.drawing.canvas.2d***. This can also be expanded using the object literal pattern to be:

```
var application = {
      utilities:{
          drawing:{
              canvas:{
                  2d:{
                          //...
                  }
              }
          }
     }
};
```

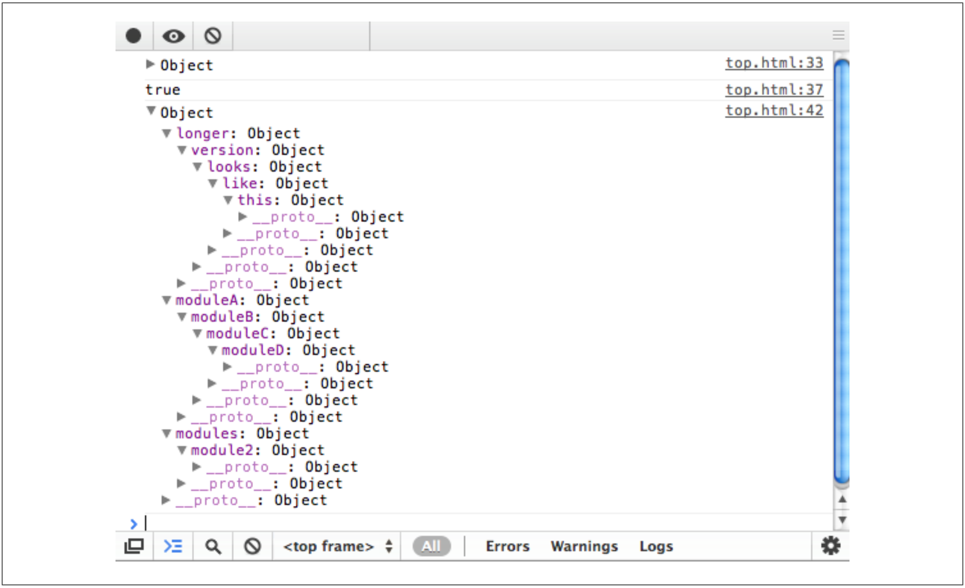
One of the obvious challenges with this pattern is that each additional layer we wish to create requires yet another object to be defined as a child of some parent in our top-level namespace. This can become particularly laborious when multiple depths are required as our application increases in complexity.

How can this problem be better solved? In JavaScript Patterns, Stoyan Stefanov presents a very clever approach for automatically defining nested namespaces under an existing global variable. He suggests a convenience method that takes a single string argument for a nest, parses this, and automatically populates our base namespace with the objects required.

The method he suggests using is the following, which I've updated it to be a generic function for easier reuse with multiple namespaces:

```
// top-level namespace being assigned an object literal
var myApp = myApp || {};

// a convenience function for parsing string namespaces and
// automatically generating nested namespaces
function extend( ns, ns_string ) {
    var parts = ns_string.split("."),
        parent = ns,
        pl;

    pl = parts.length;

    for ( var i = 0; i < pl; i++ ) {
        // create a property if it doesn't exist
        if ( typeof parent[parts[i]] === "undefined" ) {
            parent[parts[i]] = {};
        }

        parent = parent[parts[i]];
    }

    return parent;
}

// Usage:
// extend myApp with a deeply nested namespace
var mod = extend(myApp, "myApp.modules.module2");

// the correct object with nested depths is output
console.log(mod);

// minor test to check the instance of mod can also
// be used outside of the myApp namesapce as a clone
// that includes the extensions

// Outputs: true
console.log(mod == myApp.modules.module2);

// further demonstration of easier nested namespace
// assignment using extend
extend(myApp, "moduleA.moduleB.moduleC.moduleD");
extend(myApp, "longer.version.looks.like.this");
console.log(myApp);
```

Figure 13-1 shows the Chrome Developer Tools output:



```
▶ Object                                   top.html:33
true                                       top.html:37
▼ Object                                   top.html:42
  ▼ longer: Object
    ▼ version: Object
      ▼ looks: Object
        ▼ like: Object
          ▼ this: Object
            ▶ __proto__: Object
          ▶ __proto__: Object
        ▶ __proto__: Object
      ▶ __proto__: Object
    ▶ __proto__: Object
  ▼ moduleA: Object
    ▼ moduleB: Object
      ▼ moduleC: Object
        ▼ moduleD: Object
          ▶ __proto__: Object
        ▶ __proto__: Object
      ▶ __proto__: Object
    ▶ __proto__: Object
  ▼ modules: Object
    ▼ module2: Object
      ▶ __proto__: Object
    ▶ __proto__: Object
  ▶ __proto__: Object
>
```

Figure 13-1. Chrome Developer Tools output

Where one would previously have had to explicitly declare the various nests for their namespace as objects, this can now be easily achieved using a single, cleaner line of code.

Dependency Declaration Pattern

We're now going to explore a minor augmentation to the Nested Namespacing pattern which we'll refer to as the Dependency Declaration pattern. We all know that local references to objects can decrease overall lookup times, but let's apply this to name-spacing to see how it might look in practice:

```
// common approach to accessing nested namespaces
myApp.utilities.math.fibonacci( 25 );
myApp.utilities.math.sin( 56 );
myApp.utilities.drawing.plot( 98,50,60 );

// with local/cached references
var utils = myApp.utilities,
maths = utils.math,
drawing = utils.drawing;

// easier to access the namespace
maths.fibonacci( 25 );
maths.sin( 56 );
drawing.plot( 98, 50,60 );

// note that the above is particularly performant when
```

```
    // compared to hundreds or thousands of calls to nested
    // namespaces vs. a local reference to the namespace
```

Working with a local variable here is almost always faster than working with a top-level global (e.g., `myApp`). It's also both more convenient and more performant than accessing nested properties/subnamespaces on every subsequent line and can improve readability in more complex applications.

Stoyan recommends declaring localized namespaces required by a function or module at the top of our function scope (using the single-variable pattern) and calls this a Dependency Declaration pattern. One of the benefits this offers is a decrease in locating dependencies and resolving them, should we have an extendable architecture that dynamically loads modules into our namespace when required.

In my opinion, this pattern works best when working at a modular level, localizing a namespace to be used by a group of methods. Localizing namespaces on a per-function level, especially where there is significant overlap between namespace dependencies, would be something I would recommend avoiding where possible. Instead, define it further up and just have them all access the same reference.

Deep Object Extension

An alternative approach to automatic namespacing is deep object extension. Namespaces defined using object literal notation may be easily extended (or merged) with other objects (or namespaces) such that the properties and functions of both namespaces can be accessible under the same namespace post-merge.

This is something that's been made fairly easy to accomplish with modern JavaScript frameworks (e.g., see jQuery's $.extend); however, if looking to extend objects (namespaces) using vanilla JS, the following routine may be of assistance.

```
// extend.js
// Written by Andrew Dupont, optimized by Addy Osmani

function extend( destination, source ) {

    var toString = Object.prototype.toString,
        objTest = toString.call({});

    for ( var property in source ) {
        if ( source[property] && objTest === toString.call(source[property]) ) {
            destination[property] = destination[property] || {};
            extend(destination[property], source[property]);
        } else {
            destination[property] = source[property];
        }
    }
    return destination;

};
```

```
console.group( "objExtend namespacing tests" );

// define a top-level namespace for usage
var myNS = myNS || {};

// 1. extend namespace with a "utils" object
extend(myNS, {
        utils:{
        }
});

console.log( "test 1" , myNS);
// myNS.utils now exists

// 2. extend with multiple depths (namespace.hello.world.wave)
extend(myNS, {
                hello:{
                        world:{
                                wave:{
                                    test: function(){
                                        //...
                                    }
                                }
                        }
                }
});

// test direct assignment works as expected
myNS.hello.test1 = "this is a test";
myNS.hello.world.test2 = "this is another test";
console.log( "test 2", myNS );

// 3. what if myNS already contains the namespace being added
// (e.g. "library")? we want to ensure no namespaces are being
// overwritten during extension

myNS.library = {
        foo:function () {}
};

extend( myNS, {
        library:{
                bar:function(){
                    //...
                }
        }
});

// confirmed that extend is operating safely (as expected)
// myNS now also contains library.foo, library.bar
console.log( "test 3", myNS );


// 4. what if we wanted easier access to a specific namespace without having
// to type the whole namespace out each time?
```

```
var shorterAccess1 = myNS.hello.world;
shorterAccess1.test3 = "hello again";
console.log( "test 4", myNS);

//success, myApp.hello.world.test3 is now "hello again"

console.groupEnd();
```

The above implementation is not cross-browser compatible for all objects and should be considered a proof-of-concept only. One may find the Underscore.js `extend()` method a simpler, more cross-browser-friendly implementation to start with: *http://documentcloud.github.com/underscore/docs/underscore.html#section-67*. Alternatively, a version of the jQuery `$.extend()` method extracted from core can be found here: *https://github.com/addyosmani/jquery.parts*.

For developers who are going to use jQuery in their applications, one can achieve the exact same object namespace extensibility with `$.extend` as follows:

```
// top-level namespace
var myApp = myApp || {};

// directly assign a nested namespace
myApp.library = {
  foo:function(){
    //...
  }
};

// deep extend/merge this namespace with another
// to make things interesting, let's say it's a namespace
// with the same name but with a different function
// signature: $.extend( deep, target, object1, object2 )
$.extend( true, myApp, {
    library:{
        bar:function(){
            //...
        }
    }
});

console.log("test", myApp);
// myApp now contains both library.foo() and library.bar() methods
// nothing has been overwritten which is what we're hoping for.
```

For the sake of thoroughness, please see here for jQuery `$.extend` equivalents to the rest of the namespacing experiments found in this section.

Recommendation

Reviewing the namespace patterns we've explored in this section, the option that I would personally use for most larger applications is nested object namespacing with the Object Literal pattern. Where possible, I would implement this using automated nested namespacing, however, this is just a personal preference.

IIFEs and single global variables may work fine for applications in the small to medium range, however, larger code bases requiring both namespaces and deep subnamespaces require a succinct solution that promotes readability and scales. I feel this pattern achieves all of these objectives well.

I would also recommend trying out some of the suggested advanced utility methods for namespace extension, as they really can save time in the long run.

CHAPTER 14
Conclusions

That's it for this introductory adventure into the world of design patterns in JavaScript and jQuery. I hope you've found it beneficial.

Design patterns make it easy for us to build on the shoulders of developers who have defined solutions to challenging problems and architectures over a number of decades. The contents of this book should hopefully provide sufficient information to get you started using the patterns we covered in your own scripts, plug-ins, and web applications.

It's important for us to be aware of these patterns, but it's also essential to know how and when to use them. Study the pros and cons of each pattern before employing them. Take the time out to experiment with patterns to fully appreciate what they offer and make usage judgements based on a pattern's true value to your application.

If I've encouraged your interest in this area further and you would like to learn more about design patterns, there are a number of excellent titles on this area available for generic software development and of course, JavaScript.

I am happy to recommend:

- Patterns Of Enterprise Application Architecture by Martin Fowler
- JavaScript Patterns by Stoyan Stefanov

Thanks for reading *Learning JavaScript Design Patterns*. For more educational material on learning JavaScript, please feel free to read more from me on my blog at *http://addyosmani.com* or on Twitter @addyosmani.

Until next time, the very best of luck with your adventures in JavaScript!

APPENDIX

References

1. Robert C Martin, “Design Principles and Design Patterns”.
2. Ralph Johnson, “Special Issue on Patterns and Pattern Languages”, ACM.
3. Hillside Engineering Design Patterns Library.
4. Ross Harmes and Dustin Diaz, “Pro JavaScript Design Patterns”.
5. Design Pattern Definitions.
6. Patterns and Software Terminology.
7. Jeff Juday, “Reap the benefits of Design Patterns”.
8. Subramanyan, “Guhan, JavaScript Design Patterns”.
9. James Moaoriello, “What Are Design Patterns and Do I Need Them?”.
10. Alex Barnett, “Software Design Patterns”.
11. Gunni Rode, “Evaluating Software Design Patterns”.
12. SourceMaking Design Patterns.
13. “The Singleton”,Prototyp.ical.
14. Stoyan Stevanov, “JavaScript Patterns”.
15. Design Pattern Implementations in JavaScript; discussion, Stack Overflow.
16. Jared Spool, “The Elements of a Design Pattern”.
17. Examples of Practical JS Design Patterns; discussion, Stack Overflow.
18. Nicholas Zakkas, “Design Patterns in JavaScript Part 1”.
19. Design Patterns in jQuery, Stack Overflow.
20. Elyse Neilson, “Classifying Design Patterns By AntiClue”.
21. Douglas Schmidt, “Design Patterns, Pattern Languages, and Frameworks”.
22. Christian Heilmann, “Show Love To The Module Pattern”.
23. Mike G., “JavaScript Design Patterns”.
24. Anoop Mashudanan, “Software Designs Made Simple”.

25. Klaus Komenda, “JavaScript Design Patterns”.
26. Introduction to the JavaScript Module Pattern.
27. Design Patterns Explained.
28. Mixins explained.
29. Working with GoF’s Design Patterns In JavaScript.
30. Using Object.create.
31. t3knomanster, JavaScript Design Patterns.
32. Working with GoF Design Patterns In JavaScript Programming.
33. JavaScript Advantages of Object Literal, Stack Overflow.
34. Liam McLennan, “JavaScript Class Patterns”.
35. Understanding proxies in jQuery.
36. Observer Pattern Using JavaScript.
37. Speaking on the Observer pattern.
38. Singleton examples in JavaScript, Hardcode.nl.
39. Design Patterns by Gamma, Helm supplement.

Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

About the Author

Addy Osmani is a Developer Programs Engineer on the Chrome team at Google with a passion for JavaScript application architecture. He has created popular projects such as TodoMVC and contributed to other open source projects such as Modernizr and jQuery. A regular blogger, his articles have been featured numerous times in *JavaScript Weekly* and *Smashing Magazine*, to name but a few. His current projects include Yeoman (an opinionated workflow for modern web developers), Backbone Aura, and jQuery UI Bootstrap. Addy maintains his popular blog on web development at http://addyosmani.com, for those wishing to learn more about his work.

Colophon

The animal on the cover of *Learning JavaScript Design Patterns* is the cuckoo pheasant (*Dromococcyx phasianellus,* also known as the pheasant cuckoo). The pheasant cuckoo is a bird that is native to forests from the Yucatan peninsula to Brazil, and may be found as far south as Colombia. In an era when many avian species are endangered due to habitat destruction, the pheasant cuckoo has managed to hold a Conservation Status of “least concern.”

The pheasant cuckoo has a long tail and a short, dark brown crest. Its diet consists of insects, which it catches by making rattling sounds with its feathers and clapping its bill, then running several steps forward and pecking at the ground. Although it is an insectivore, it may also feed on small lizards and nestlings.

Like many other cuckoos, the pheasant cuckoo lays its eggs in another bird’s nest. When the eggs hatch, the adoptive mother will recognize the cuckoo’s offspring as her own, and the hatchlings will imprint on their adoptive mother. Acting on instinct, cuckoo hatchlings will push the host parent’s eggs out of the nest to make room for themselves. Unlike the European cuckoo, however, the pheasant cuckoo is not an obligate brood parasite; it still has the ability to construct nests of its own.

The cover image is a loose plate engraving, origin unknown. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont’s TheSansMonoCondensed.