Essential JavaScript Design Patterns

by A BOOK BY ADDY OSMANI

Preface

Design patterns are reusable solutions to commonly occurring problems in software design. They are both exciting and a 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.

Acknowledgements

I will always be grateful for the 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 reviewer's tweets and blogs are also a regular source of both ideas and inspiration and I wholeheartedly recommend checking them out.

I would also like to thank 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.

Credits

Whilst some 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

Whilst 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 langage, 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

Table Of Contents

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 the chapter 'What is a Pattern?' to continue

reading.

Design patterns can be traced back to the early work of a civil engineer 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 Sarah Ishikawra 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 twenty-three core Object-Oriented design patterns frequently used around the world today. We will be covering these patterns in more detail in the section ‘Categories of Design Patterns’.

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, but also to abstracted libraries such as jQuery or Dojo as well. Before we begin, let’s look at the exact definition of a ‘pattern’ in software design.

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-powered applications. Another way of

looking at patterns are as templates for how you solve problems - ones which 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:

1. **Patterns are proven solutions:** They provide solid approaches to solving issues in software development using proven solutions that reflect the experience and insights the developers that helped define and improve them bring to the pattern.
2. **Patterns can be easily re-used:** A pattern usually reflects an out of the box solution that can be adapted to suit your own needs. This feature makes them quite robust.
3. **Patterns can be expressive:** When you 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** an exact solution. 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 so the need to refactor it for cleanliness purposes in the future.

- **Patterns can provide generalized solutions which are 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) you are working with, design patterns can be applied to improve the structure of your code.
- **Certain patterns can actually decrease the overall file-size footprint of your 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 your codebase can be decreased.
- **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 whilst 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 everyday

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

If we imagine that we have a script where for each DOM element on a page with class "foo" we want to increment a counter, what's the simplest efficient way to query for the list we need?. Well, there are a few different ways this problem could be tackled:

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

So, which of these is the fastest?. You might be interested to know that it's actually number 3 by a factor of 8-10 times the alternatives. In a real-world application however, 3. will not work in versions of Internet Explorer below 9 and thus it's necessary to use 1. where 3. isn't supported.

Developers using jQuery don't have to worry about this problem, as it's luckily abstraced away for us. The library opts for the most optimal approach to selecting elements depending on what your browser supports.

It internally uses a number of different **design patterns**, the most frequent one being a facade, which provides a simple set of interfaces to a more complex body of code. We're probably all familiar with `$(elem)` (yes, it's a facade!), which is a lot easier to use than having to manually normalize cross-browser differences.

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

'Pattern'-ity Testing, Proto-Patterns & 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 weary of something claiming to be one unless it has been heavily vetted. Even if something is presented to us which *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 both be a process and a 'thing'. This definition is obtuse on purpose as he follows by saying that it is the process should create the 'thing'. This is a reason why patterns generally focus on addressing a visually identifiable structure i.e you should be able to visually depict (or draw) a picture representing the structure that placing the pattern into practice results in.

In studying design patterns, you may come across the term 'proto-pattern' quite frequently. What is this? Well, a pattern that has not yet been known to pass the 'pattern'-ity tests is usually referred to as a proto-pattern. Proto-patterns may result from the work of someone that has established a particular solution that is worthy of sharing with the community, but may not have yet had the opportunity to have been vetted heavily due to it's 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 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.
- **The solution to this problem cannot be obvious** : You can often find that problem-solving techniques 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.
- **The concept described must have been proven**: 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.
- **It must describe 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 it's relationship to code.

You wouldn't be blamed for thinking that a proto-pattern which doesn't meet the guidelines is worth learning from at all, but 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 you with future projects. Use best judgment with the

above list in mind and you'll be fine in your selection process.

One of the additional requirements for a pattern to be valid is that they 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:

1. **Fitness of purpose** - how is the pattern considered successful?
2. **Usefulness** - why is the pattern considered successful?
3. **Applicability** - is the design worthy of being a pattern because it has wider applicability? If so, this needs to be explained.When reviewing or defining a pattern, it is important to keep the above in mind.

The Structure Of A Design Pattern

When studying design patterns, you may wonder what teams that create them have to put in their design pattern descriptions. Every pattern has to initially be formulated in a form of a **rule** that establishes a relationship between a **context**, a system of **forces** that arises in that context and a **configuration** that allows these forces to resolve themselves in context.

I find that a lot of the information available out there about the structure of a good pattern can be condensed down to something more easily digestible.With this in mind, lets now take a look at a summary of the component elements for a design pattern.

A design pattern must have a:

- **Pattern Name** and a **description**
- **Context Outline** – the contexts in which the pattern is effective in responding to the 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** – a visual representation of classes in the pattern (eg. a diagram))
- **Examples** – an implementation of the pattern in a minimal form
- **Co-requisites** – 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 or your company ever consider working on your own pattern, remember that although they 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 than starting afresh.

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 you a number of useful benefits. For starters, you can gain a deeper appreciation for the reasoning behind a pattern being needed but can also learn how to tell if a pattern (or proto-pattern) is up to scratch when reviewing it for your own needs.

Writing good patterns is a challenging task. Patterns not only need to provide a substantial quantity of reference material for end-users (such as the items found in the *structure* section above), but they also need to be able to almost tell a 'story' that describes the experience they are trying to convey. If you've already read the previous section on 'what' a pattern is, you may think that this in itself should help you identify patterns when you see them in the wild. This is actually quite the opposite - you can't always tell if a piece of code you're inspecting follows a pattern.

When looking at a body of code that you think may be using a pattern, you might write down some of the aspects of the code that you believe falls under a particular existing pattern, but it may not be a one at all. In many cases of pattern-analysis you'll find that you'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're 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 books and take in what's meaningful to you - this will help you accomplish the goals you've got of designing the pattern you want to achieve. You'll probably also want to examine the structure and semantics of existing patterns - this can be begun 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 you've exposed yourself to a wealth of information on pattern literature, you may wish to begin your pattern using an *existing* format and see if you can brainstorm new ideas for improving it or integrating your ideas in there. An example of someone that did this quite recently is JavaScript developer Christian Heilmann, who took an existing pattern called the *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).

If you would like to try your hand at writing a design pattern (even if just for the learning experience of going through the process), the tips I have for doing so would be as follows:

- **Bear in mind practicability**: Ensure that your pattern describes proven solutions to recurring problems rather than just speculative solutions which haven't been qualified.
- **Ensure that you draw upon best practices:** The design decisions you make should be based on principles you derive from an understanding of best practices.
- **Your 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 behaviour 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 your design is strong enough to have broad useful applicability, it has a chance of being recognized as a proper pattern
- **Know the differences between patterns and design**: A design pattern generally draws from proven best practice and serves as a model for a designer to create a solution. *The role of the pattern is to give designers guidance to make the best design choices so they can cater to the needs of their users.*
- **Your pattern needs to have 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 your 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 if writing a pattern you cover the widest possible areas of application and you should be fine. 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.

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 which 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 life-cycle begins with construction however once you've got the initial release done, 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 have 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 the way in which design patterns provide us with a way 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 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.
- Prototyping against the Object object (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 you're on, whilst document.createElement does not. You can see here 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 you are able to recognize such anti-patterns, you will be able to refactor your code to negate them so that the overall quality of your solutions improves instantly.

Categories Of Design Pattern

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 the situation you are working in. The basic approach to object creation might otherwise lead to added complexity in a project whilst creational patterns aim to solve this problem by *controlling* the creation of such objects.

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

Structural Design Patterns

Structural patterns focus on the composition of classes and objects. Structural 'class' creation patterns use inheritance to compose interfaces whilst 'object' patterns define methods to create objects to obtain new functionality.

Patterns that fall under this category include: Decorator, Facade, Composite, Adapter and Bridge

Behavioral Design Patterns

The main focus behind this category of patterns is the communication between a class's objects. By specifically targeting this problem, these patterns are able to increase the flexibility in carrying out this communication.

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

In my early experiences of learning about design patterns, I personally found the following table 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 class-less language, however classes can be simulated using functions.

The most common approach to achieving this is by defining a JavaScript function where we then create an object using the `new` keyword. `this` can be used to help 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.

Creational	Based on the concept of creating an object.
Class	
Factory Method	This 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	Match interfaces of different classes therefore classes can work together despite incompatible interfaces
Object	
Adapter	Match interfaces of different classes therefore 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 which makes the total object more than just the sum of its parts.
Decorator	Dynamically add 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 place holder 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 grammer of the intended language.
Template Method	Creates the shell of an algorithm in a method, then defer 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	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
Iterator	Sequentially access 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	Capture 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	Alter 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

An Introduction To Design Patterns

We are now going to explore JavaScript implementations of a number of both classical and modern design patterns. This section of the book will cover an introduction to these patterns, whilst the next section will focus on looking at some select patterns in greater detail.

A common question developers regularly ask is what the 'ideal' set of patterns they should be using are. There isn't a singular answer to this question, but with the aid of what you'll learn in this book, you will hopefully be able to use your best judgement to select the right patterns to best suit your project's needs.

The patterns we will be exploring in this section are the:

The Creational Pattern

The Creational pattern is the basis for a number of the other design patterns we'll be looking at in this section and is probably the easiest to understand. As you may guess, the creational pattern deals with the idea of *creating* new things, specifically new objects. In JavaScript, the common way of creating new objects (collections of name/value) pairs is as follows:

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

```
var newObject = {}; // or

var newObject = Object.create(null); // or

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 a number of ways in which keys and values can then be assigned to an object including:

```
newObject.someKey = 'Hello World';
newObject['someKey'] = 'Hello World';

// which can be accessed in a similar fashion
var key = newObject.someKey; //or
var key = newObject['someKey'];
```

We can also define new properties on objects as follows, should we require more granular configuration capabilities:

```
// First, define a new Object 'man'
var man = Object.create(null);

// Next let's create a configuration object containing properties
// Properties can be writable, enumerable and configurable
var config = {
  writable: true,
  enumerable: true,
  configurable: true
};

// Typically one would use Object.defineProperty() to add new
// properties. For convenience we will use a short-hand version:

var defineProp = function ( obj, key, value ){
  config.value = value;
```

```
  Object.defineProperty(obj, key, config);
}

defineProp( man, 'car',  'Delorean' );
defineProp( man, 'dob', '1981' );
defineProp( man, 'beard', false );
```

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

```
var driver = Object.create( man );
defineProp (driver, 'topSpeed', '100mph');
driver.topSpeed // 100mph
```

Thanks to Yehuda Katz for the less verbose version presented above.

The Constructor Pattern

The phrase ‘constructor’ is familiar to most developers, however if you’re a beginner it can be useful to review what a constructor is before we get into talking about a pattern dedicated to it.

Constructors are used to create specific types of objects - they both prepare the object for use and can also accept parameters which the constructor uses to set the values of member variables when the object is first created. The idea that a constructor is a paradigm can be found in the majority of programming languages, including JavaScript. You’re also able to define custom constructors that define properties and methods for your own types of objects.

Basic Constructors

In JavaScript, constructor functions are generally considered a reasonable way to implement instances. As we saw earlier, JavaScript doesn't support the concept of classes but it does support special constructor functions. By simply prefixing a call to a constructor function with the keyword 'new', you can tell JavaScript you would like 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. Again, a very basic constructor may be:

```
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";
   };
}

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

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 the new objects created using the Car constructor. This isn't very optimal as the function should ideally be shared between all of the instances of the Car type.

Constructors With Prototypes

Functions in JavaScript have a property called a prototype. When you 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 which 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";
};

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

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

Here, a single instance of toString() will now be shared between all of the Car objects.

Note: Douglas Crockford recommends capitalizing your constructor functions so that it is easier to distinguish between them and normal functions.

The Singleton Pattern

In conventional software engineering, 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.

The singleton pattern is thus known because traditionally, it restricts instantiation of a class to a single object. With JavaScript, singletons serve as a namespace provider which isolate implementation code from the global namespace so-as to provide a single point of access for functions.

The singleton doesn't provide a way for code that doesn't know about a previous reference to the singleton to easily retrieve it - it is not the object or '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.

Singletons in JavaScript can take on a number of different forms and researching this pattern online is likely to result in at least 10 different variations. In its simplest form, a singleton in JS can be an object literal grouped together with its related methods and properties as follows:

```
var mySingleton = {
  property1: "something",

  property2: "something else",

  method1:function(){
    console.log('hello world');
  }

};
```

If you wished to extend this further, you could add your own private members and methods to the singleton by encapsulating variable and function declarations inside a closure. Exposing only those which you wish to make public is quite straight-forward from that point as demonstrated below:

```
var mySingleton = function(){

  // here are our private methods and variables
  var privateVariable = 'something private';
  function showPrivate(){
    console.log( privateVariable );
  }

  // public variables and methods (which can access
  // private variables and methods )
  return {

    publicMethod:function(){
      showPrivate();
    },

    publicVar:'the public can see this!'

  };
};

var single = mySingleton();
single.publicMethod();  // logs 'something private'
console.log( single.publicVar ); // logs 'the public can see this!'
```

The above example is great, but let's next consider a situation where you only want to instantiate the singleton when it's needed. To save on resources, you can place the instantiation code inside another constructor function as follows:

```
var Singleton = (function(){
  var instantiated;

  function init (){
    // singleton here
```

```
    return {
      publicMethod: function(){
        console.log( 'hello world' );
      },
      publicProperty: 'test'
    };
  }

  return {
    getInstance: function(){
      if ( !instantiated ){
        instantiated = init();
      }
      return instantiated;
    }
  };
})();

// calling public methods is then as easy as:
Singleton.getInstance().publicMethod();
```

So, where else is the singleton pattern useful in practice?. Well, it's quite useful when exactly one object is needed to coordinate patterns across the system. Here's one last example of the singleton pattern being used:

```
var SingletonTester = (function(){

  // args: an object containing arguments for the singleton
  function Singleton( args ) {

   // set args variable to args passed or empty object if none provided.
    var args = args || {};
    //set the name parameter
    this.name = 'SingletonTester';
    //set the value of pointX
    this.pointX = args.pointX || 6; //get parameter from arguments or set def
ault
    //set the value of pointY
    this.pointY = args.pointY || 10;
```

```
  }

 // this is our instance holder
  var instance;

 // this is an emulation of static variables and methods
  var _static = {
    name: 'SingletonTester',
   // This is a method for getting an instance

   // It returns a singleton instance of a singleton object
    getInstance: function ( args ){
      if (instance === undefined) {
        instance = new Singleton( args );
      }
      return instance;
    }
  };
  return _static;
})();

var singletonTest = SingletonTester.getInstance({pointX: 5});
console.log(singletonTest.pointX); // outputs 5
```

The Module Pattern

Let's now look at the popular *module* pattern. Note that we'll be covering this pattern in greater detail in the next section of the book, but a basic introduction to it will be given in this chapter.

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 your function names conflicting with other functions defined in additional scripts on the page.

JavaScript as a language doesn't have access modifiers that would allow us to implement true

privacy, but for the purposes of most use cases, simulated privacy should work fine.

Exploring the concept of public and private methods further, the module pattern pattern allows us to have particular methods and variables which are only accessible from within the module, meaning that you have a level of shielding from external entities accessing this 'hidden' information.

Let's begin looking at an implementation of the module pattern by creating a module which is self-contained. 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 (eg. 'testModule').

```
var testModule = (function(){
  var counter = 0;
  return {
    incrementCounter: function() {
      return counter++;
    },
    resetCounter: function() {
      console.log('counter value prior to reset:' + counter);
      counter = 0;
    }
  };
})();

// test
testModule.incrementCounter();
testModule.resetCounter();
```

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

```
var myNamespace = (function(){
```

```
  var myPrivateVar = 0;
  var myPrivateMethod = function( someText ){
    console.log(someText);
  };

  return {

    myPublicVar: "foo",

    myPublicFunction: function(bar){
      myPrivateVar++;
      myPrivateMethod(bar);
    }
  };

})();
```

A piece of trivia is that the module pattern was originally defined by Douglas Crockford (famous for his book 'JavaScript: The Good Parts, and more), although it is likely that variations of this pattern were used long before this. Another piece of trivia is that if you've ever played with Yahoo's YUI library, some of its features may appear quite familiar and the reason for this is that the module pattern was a strong influence for YUI when creating their components.

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. Secondly, it supports private data - so, in the module pattern, public parts of your code are able to touch the private parts, however the outside world is unable to touch the class's private parts (no laughing! Oh, and thanks to David Engfer for the joke).

Disadvantages: The disadvantages of the module pattern are that as you access both public and private members differently, when you wish to change visibility, you actually have to make changes to each place the member was used. You 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 your application. Here's a final module pattern example:

```
var someModule = (function(){

  // private attributes
  var privateVar = 5;

  // private methods
  var privateMethod = function(){
     return 'Private Test';
  };

  return {
    // public attributes
    publicVar: 10,
    // public methods
    publicMethod: function(){
      return ' Followed By Public Test ';
    },

    // let's access the private members
    getData: function(){
      return privateMethod() + this.publicMethod() + privateVar;
    }
  }
})(); //the parens here cause the anonymous function to execute and return

someModule.getData();
```

To continue reading more about the module pattern, I strongly recommend Ben Cherry's JavaScript Module Pattern 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 (now at Mozilla) was frustrated with the fact that if you had to repeat the name of the main object when you wanted to call one public method from another or access public variables. He also disliked the Module pattern's requirement for having to switch to object literal notation for the things you wished to make

public.

The result of his efforts were an updated pattern where you would simply define all of your functions and variables in the private scope and return an anonymous object at the end of the module along with pointers to both the private variables and functions you wished to reveal as public.

Once again, you're probably wondering what the benefits of this approach are. The Reveling Module Pattern allows the syntax of your script to be fairly consistent - it also makes it very clear at the end which of your functions and variables may be accessed publicly, something that is quite useful. In addition, you are also able to reveal private functions with more specific names if you wish.

An example of how to use the revealing module pattern can be found below:

```
 var myRevealingModule = (function(){

    var name = 'John Smith';
    var age = 40;

    function updatePerson(){
      name = 'John Smith Updated';
    }
    function setPerson () {
       name = 'John Smith Set';
    }
    function getPerson () {
       return name;
    }
    return {
        set: setPerson,
        get: getPerson
    };
}());

// Sample usage:
myRevealingModule.get();
```

The Observer Pattern

The Observer pattern (also known as the Publish/Subscribe model) is a design pattern which allows an object (known as an observer) to watch another object (the subject) where the pattern provides a means for the subject and observer to form a publish-subscribe relationship. It is regularly used when we wish to decouple the different parts of an application from one another.

Note that this is another pattern we'll be looking at in greater detail in the next section of the book.

Observers are able to register (subscribe) to receive notifications from the subject when something interesting happens. When the subject needs to notify observers about interesting events, it broadcasts (publishes) a notification of these events to each observer (which can include data related to the event). .

The motivation behind using the observer pattern is where you 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. Another use case is where abstractions have more than one aspect, where one depends on the other. The encapsulation of these aspects in separate objects allows the variation and re-use of the objects independently.

Advantages using the observer pattern include:

- Support for simple broadcast communication. Notifications are broadcast automatically to all objects that have subscribed.
- Dynamic relationships may exist between subjects and observers which can be easily established on page load. This provides a great deal of flexibility.
- Abstract coupling between subjects and observers where each can be extended and re-used individually.

Disadvantages

A draw-back of the pattern is that observers are ignorant to the existence of each other and are blind to the cost of switching in subject. Due to the dynamic relationship between subjects and observers the update dependency can be difficult to track.

Let us now take a look at an example of the observer pattern implemented in JavaScript. The following demo is a minimalist version of Pub/Sub I released on GitHub under a project called pubsubz. Sample usage of this implementation can be seen shortly.

Observer 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;
        }

        setTimeout(function() {
            var subscribers = topics[topic],
                len = subscribers ? subscribers.length : 0;

            while (len--) {
                subscribers[len].func(topic, args);
            }
        }, 0);

        return true;

    };

    // 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
```

Observing and broadcasting

We can now use the implementation to publish and subscribe to events of interest as follows:

```
var testSubscriber = function( topics , data ){
    console.log( topics + ": " + data );
};

// Publishers are in charge of "publishing" notifications about events

pubsub.publish( 'example1', 'hello world!' );
pubsub.publish( 'example1', ['test','a','b','c'] );
pubsub.publish( 'example1', [{'color':'blue'},{'text':'hello'}] );

// Subscribers basically "subscribe" (or listen)
// And once they've been "notified" their callback functions are invoked
var testSubscription = pubsub.subscribe( 'example1', testSubscriber );

// Unsubscribe if you no longer wish to be notified

setTimeout(function(){
    pubsub.unsubscribe( testSubscription );
}, 0);

pubsub.publish( 'example1', 'hello again! (this will fail)' );
```

A jsFiddle version of this example can be found at http://jsfiddle.net/LxPrq/

Note: If you are interested in a pub/sub pattern implementation using jQuery, I recommend Ben Alman's GitHub Gist for an example of how to achieve this.

The dictionary refers to a Mediator as 'a neutral party who assists in negotiations and conflict resolution'.

In software engineering, a Mediator is a behavioural 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 may have too many direct relationships between modules (colleagues), it may be time to have a central point of control that modules communicate through instead. The Mediator promotes loose coupling by ensuring that instead of modules referring to each other explicitly, their interaction is handled through this central point.

If you would prefer an analogy, consider a typical airport traffic control system. A tower (Mediator) handles what planes (modules) can take off and land because all communications are done 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 a mediator plays in software design.

In real-world terms, a mediator encapsulates how disparate modules interact with each other by acting as an intermediary. At it's most basic, a mediator could be implemented as a central base for accessing functionality as follows:

```
// Our app namespace can act as a mediator
var app = app || {};

// Communicate through the mediator for Ajax requests
app.sendRequest = function ( options ) {
    return $.ajax($.extend({}, options);
}

// When a request for a URL resolves, do something with the view
app.populateView = function( url, view ){
  $.when(app.sendRequest({url: url, method: 'GET'})
     .then(function(){
         //populate the view
     });
}

// Empty a view of any content it may contain
app.resetView = function( view ){
   view.html('');
}
```

That said, in the JavaScript world it's become quite common for the Mediator to act as a messaging bus on top of the Observer-pattern. Rather than modules calling a Publish/Subscribe implementation, they'll use a Mediator with these capabilities built in instead. A possible implementation of this (based on work by Ryan Florence) could look as follows:

```
var mediator = (function(){
    // Subscribe to an event, supply a callback to be executed
    // when that event is broadcast
    var subscribe = function(channel, fn){
        if (!mediator.channels[channel]) mediator.channels[channel] = [];
        mediator.channels[channel].push({ context: this, callback: fn });
        return this;
    },

    // Publish/broadcast an event to the rest of the application
```

```
    publish = function(channel){
        if (!mediator.channels[channel]) return false;
        var args = Array.prototype.slice.call(arguments, 1);
        for (var i = 0, l = mediator.channels[channel].length; i
```

Here are two sample uses of the implementation from above. It's effectively centralized Publish/Subscribe where a mediated implementation of the Observer pattern is used:

```
(function( m ){

    function initialize(){

      // Set a default value for 'person'
      var person = "tim";

      // Subscribe to an event called 'nameChange' with
      // a callback function which will log the original
      // person's name and (if everything works) the new
      // name

      m.subscribe('nameChange', function( arg ){
          console.log( person ); // tim
          person = arg;
          console.log( person ); // david
     });
    }

    function updateName(){
      // Publish/Broadcast the 'nameChange' event with the new data
      m.publish( 'nameChange', 'david' );
    }

})( mediator );
```

Advantages & Disadvantages

The benefits of the Mediator pattern are that it simplifies object interaction and can aid with decoupling those using it as a communication hub. In the above example, rather than using the Observer pattern to explicitly set many-to-many listeners and events, a Mediator allows you to broadcast events globally between subscribers and publishers. Broadcasted events can be handled by any number of modules at once and a mediator can used for a number of other purposes such as permissions management, given that it can control what messages can be subscribed to and which can be broadcast.

Perhaps the biggest downside of using the Mediator pattern is that it can introduce a single point of failure. Placing a Mediator between modules can also 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 your 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 which can work very well if implemented correctly.

Mediator Vs. 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.*"

The Mediator pattern centralizes rather than simply just distributing. It places the responsibility for maintaining a constraint squarely in the mediator.

Mediator Vs. 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 s

ubtle differences.

The Mediator centralizes communication between modules where it's explicitly referenced by these modules. In a sense this is multidirectional. The Facade however 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 refer to the prototype pattern as one which 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 where we create objects which 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 lower down), then each object created by that same constructor will also have this same property.

Looking at the definitions for the prototype pattern in existing literature non-specific to JavaScript, you *may* find references to concepts outside the scope of the language such as classes. 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 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 also come with a performance boost as well: when defining a function in an object, they're all created by reference (s

o all child objects point to the same function) instead of creating their own individua l copies.

For those interested, real prototypal inheritance, as defined in the ECMAScript 5 stand ard, requires the use of `Object.create` which has only become broadly native at the time o f writing. `Object.create` creates an object which has a specified prototype and which opti onally contains specified properties (i.e `Object.create(prototype, optionalDescriptorObjects)`). We can also see this being demonstrated in the example below:

```
// No need for capitalization as it's not a constructor
var someCar = {
  drive: function() {},
  name: 'Mazda 3'
};

// Use Object.create to generate a new car
var anotherCar = Object.create( someCar );
anotherCar.name = 'Toyota Camry';
```

Object.create allows you to easily implement advanced concepts such as differential inh eritance where objects are able to directly inherit from other objects. With Object.cre ate you're also able to initialise 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(),
    enumerable: true // writable:false, configurable:false by default
  },
  'model':{
    value: 'Ford',
    enumerable: true
  }
});
```

Here the properties can be initialized on the second argument of Object.create using an object literal using the syntax similar to that used by the Object.defineProperties and Object.defineProperty methods. It allows you to set the property attributes such as enumerable, writable or configurable.

If you wish to implement the prototype pattern without directly using Object.create, you 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();
```

The Command Pattern

The command pattern aims to encapsulate method invocation, requests or operations into a single object and gives you the ability to both parameterize and pass method calls ar

ound that can be executed at your discretion. In addition, it enables you to decouple objects invoking the action from the objects which implement them, giving you a greater degree of overall flexibility in swapping out concrete 'classes'.

If you haven't come across concrete classes before, they 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 which implements the missing functionality is called a concrete class (you may find these concepts familiar if you're read about the Decorator or Prototype patterns).

The main idea behind the command pattern is that it provides you 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 + ' ) ';
      }

    };

})();
```

Now taking a look at the above code, we could easily execute our manager commands by directly invoking the methods, however in some situations we don't expect to invoke the inner methods inside the object directly.

The reason for this is that we don't want to increase the dependencies amongst objects i.e if the core login behind the CarManager changes, all our methods that carry out the processing with the manager have to be modified in the mean time. This would effectively go against the OOP methodology of loosely coupling objects as much as possible which we want to avoid.

Let's now expand on our CarManager so that our application of the command pattern results in the following: accept any process requests from the CarManager object where the contents of the request include the model and car ID.

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

```
CarManager.execute({commandType: "buyVehicle", operand1: 'Ford Escort', operand2: '453543'});
```

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

```
CarManager.execute = function( command ){
  return CarManager[command.request](command.model,command.carID);
};
```

Our final sample calls would thus look as follows:

```
CarManager.execute({request: "arrangeViewing", model: 'Ferrari', carID: '145523'});
CarManager.execute({request: "requestInfo", model: 'Ford Mondeo', carID: '543434'});
CarManager.execute({request: "requestInfo", model: 'Ford Escort', carID: '543434'});
CarManager.execute({request: "buyVehicle", model: 'Ford Escort', carID: '543434'});
```

The DRY Pattern

Disclaimer: DRY is essentially a way of thinking and many patterns aim to achieve a level of DRY-ness with their design. In this section we'll be covering what it means for code to be DRY but also covering the DRY design pattern based on these same concepts.

A challenge that developers writing large applications frequently have is writing similar code multiple times. Sometimes this occurs because your script or application may have multiple similar ways of performing something. Repetitive code writing generally reduces productivity and leaves you open to having to re-write code you've already written similar times before, thus leaving you with less time to add in new functionality.

DRY (don't repeat yourself) was created to simplify this – it's based on the idea that each part of your code should ideally only have one representation of each piece of knowledge in it that applies to your system. The key concept to take away here is that if you have code that performs a specific task, you shouldn't write that code multiple times through your applications or scripts.

When DRY is applied successfully, the modification of any element in the system doesn't change other logically-unrelated elements. Elements in your code that are logically related change uniformly and are thus kept in sync.

As other patterns covered display aspects of DRY-ness with JavaScript, let's take a look at how to write DRY code using jQuery. Note that where jQuery is used, you can easily substitute selections using vanilla JavaScript because jQuery is just JavaScript at an abstracted level.

Non-DRY

```
// Let's store some defaults about a car for reference
var defaultSettings = {};
defaultSettings['carModel']   = 'Mercedes';
defaultSettings['carYear']    = 2010;
defaultSettings['carMiles']   = 5000;
defaultSettings['carTint']    = 'Metallic Blue';

// Let's do something with this data if a checkbox is clicked
$('.someCheckbox').click(function(){

 if ( this.checked ){

    $('#input_carModel').val(activeSettings.carModel);
    $('#input_carYear').val(activeSettings.carYear);
    $('#input_carMiles').val(activeSettings.carMiles);
    $('#input_carTint').val(activeSettings.carTint);

 } else {

    $('#input_carModel').val('');
    $('#input_carYear').val('');
    $('#input_carMiles').val('');
    $('#input_carTint').val('');
 }
});
```

DRY

```
$('.someCheckbox').click(function(){
    var checked = this.checked,
    fields = ['carModel', 'carYear', 'carMiles', 'carTint'];
    /*
        What are we repeating?
        1. input_ precedes each field name
        2. accessing the same array for settings
        3. repeating value resets

        What can we do?
        1. programmatically generate the field names
        2. access array by key
        3. merge this call using terse coding (ie. if checked,
            set a value, otherwise don't)
  */
  $.each(fields, function(i,key){
    $('#input_' + key).val(checked ? defaultSettings[key] : '');
  });
});
```

The Facade Pattern

When we put up a facade, we present an outward appearance to the world which 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. The facade 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 which almost always improves usability.

Facades are a structural pattern which can often be seen in JavaScript libraries like 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 rather than the subsystem behind the scenes. Whenever you're using jQuery's `$(el).css()` or `$(el).animate()` methods, you're actually using a facade – the simpler public interface that avoids you having to manually call the many internal methods in jQuery core required to get some behaviour working.

The facade pattern both simplifies the interface of a class and it also decouples the class from the code that utilizes 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 size-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 utilizing a facade to simplify an interface for listening to events cross-browser. We do this by creating a common method that can be used in one’s code which does t

he task of checking for the existence of features so that it can provide a safe and cro ss-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, t his is actually being 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, where the rest of the world simply uses the limite d 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 you can see below, our instance of the module patterns contains a number of methods which 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();
            }
        }
    }
}());


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

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

The Factory Pattern

Similar to other creational patterns, the Factory Pattern deals with the problem of creating objects (which we can think of as ‘factory products’) without the need to specify the exact class of object being created.

Specifically, the Factory Pattern suggests defining an interface for creating an object where you allow the subclasses to decide which class to instantiate. This pattern handles the problem by defining a completely separate method for the creation of objects and which sub-classes are able to override so they can specify the ‘type’ of factory product that will be created.

This can come in quite useful, in particular if the creation process involved is quite complex. eg. if it strongly depends on the settings in configuration files.

You can often find factory methods in frameworks where the code for a library may need to create objects of particular types which may be subclassed by scripts using the frameworks.

In our example, let’s take the code used in the original Constructor pattern example and see what this would look like were we to optimize it using the Factory Pattern:

```
var Car = (function() {
   var Car = function ( model, year, miles ){
       this.model = model;
       this.year   = year;
       this.miles = miles;
   };

   return function ( model, year, miles ) {
       return new Car( model, year, miles );
   };

})();

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

/*
These are also valid:
var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );
*/
```

```
When To Use This Pattern
```

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

- When your object's setup requires a high level of complexity
- When you need to generate different instances depending on the environment
- When you're working with many small objects that share the same properties

When Not To Use This Pattern

It's generally a good practice to not use the factory pattern in every situation as it can easily add an unnecessarily additional aspect of complexity to your code. It can also make some tests more difficult to run.

The Mixin Pattern

In traditional object-oriented programming languages, **mixins** are classes which provide the functionality to be inherited by a subclass. Inheriting from mixins are a means of collecting functionality and classes may inherit functionality from multiple mixins through multiple inheritance.

In the following example, we have a Car defined without any methods. We also have a constructor called 'Mixin'. What we're going to do is augment the Car so it has access to the methods within the Mixin.This code demonstrates how with JavaScript you can augment a constructor to have a particular method without using the typical inheritance methods or duplicating code for each constructor function you have.

```
// Car
var Car = function( settings ){
  this.model = settings.model || 'no model provided';
  this.colour = settings.colour || 'no colour provided';
};

// Mixin
var Mixin = function(){};
Mixin.prototype = {
```

```
  driveForward: function(){
    console.log('drive forward');
  },
  driveBackward: function(){
    console.log('drive backward');
  }
};


// Augment existing 'class' 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[argument
s[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 ( !receivingClass.prototype[methodName] ) {
        receivingClass.prototype[methodName] = givingClass.prototype[methodNa
me];
      }
    }
  }
}


// Augment the Car have the methods 'driveForward' and 'driveBackward'*/
augment( Car, Mixin,'driveForward','driveBackward' );

// Create a new Car
var vehicle = new Car({model:'Ford Escort', colour:'blue'});

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

The Decorator Pattern

The Decorator pattern is an alternative to creating subclasses. This pattern can be used to wrap objects within another object of the same interface and allows you to both add behaviour to methods and also pass the method call to the original object (i.e the constructor of the decorator).

The decorator pattern is often used when you need to keep adding new functionality to overridden methods. This can be achieved by stacking multiple decorators on top of one another.

What is the main benefit of using a decorator pattern? Well, if we examine our first definition, we mentioned that decorators are an alternative to subclassing. When a script is being run, subclassing adds behaviour that affects all the instances of the original class, whilst decorating does not. It instead can add new behaviour for individual objects, which can be of benefit depending on the application in question. Let's take a look at some code that implements the decorator pattern:

```
// This is the 'class' we're going to decorate
function Macbook(){
      this.cost = function(){
      return 1000;
     };
}

function Memory( macbook ){
    this.cost = function(){
     return macbook.cost() + 75;
  };
}

function BlurayDrive( macbook ){
   this.cost = function(){
     return macbook.cost() + 300;
  };
}


function Insurance( macbook ){
   this.cost = function(){
     return macbook.cost() + 250;
```

```
  };
}


// Sample usage
var myMacbook = new Insurance(new BlurayDrive(new Memory(new Macbook())));
console.log( myMacbook.cost() );
```

Here's another decorator example where when we invoke performTask on the decorator object, it both performs some behaviour and invokes performTask on the underlying object.

```
function ConcreteClass(){
  this.performTask = function(){
    this.preTask();
    console.log('doing something');
    this.postTask();
  };
}

function AbstractDecorator( decorated ){
  this.performTask = function()
  {
    decorated.performTask();
  };
}

function ConcreteDecoratorClass( decorated ){
  this.base = AbstractDecorator;
  this.base(decorated);

  this.preTask = function(){
    console.log('pre-calling..');
  };

  this.postTask = function(){
    console.log('post-calling..');
  };

}
```

```
var concrete = new ConcreteClass();
var decorator1 = new ConcreteDecoratorClass(concrete);
var decorator2 = new ConcreteDecoratorClass(decorator1);
decorator2.performTask();
```

Patterns In Greater Detail

As a beginner, you should hopefully now have a basic understanding of many of the commonly used design patterns in JavaScript (as well as some which are less frequently implemented). In this next section, we're going to explore a selection of the patterns we've already reviewed in greater detail.

The Observer (Pub/Sub) pattern

As we saw earlier in the book, the general idea behind the Observer pattern is the promotion of loose coupling. Rather than single objects calling on the methods of other objects directly, they instead subscribes to a specific task or activity of another object and are notified when it occurs. Observers are also called Subscribers and we refer to the object being observed as the Publisher (or the subject). Publishers notify subscribers when events occur.

When objects are no longer interested in being notified by the subject they are registered with, they can unregister (or unsubscribe) themselves. The subject will then in turn remove them from the observer collection.

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.'

Basically, the pattern describes subjects and observers forming a publish-subscribe relationship. Unlimited numbers of objects may observe events in the subject by registering themselves. Once registered to particular events, the subject will notify all observers when the event has been fired.

Advantages

Arguably, the largest benefit of using pub/sub is the ability to break down our applications into smaller, more loosely coupled modules, which can also improve general manageability.

Pub/sub is also a pattern that encourages us to think hard about the relationships between different parts of your application, identifying what layers need to observe or listen for behaviour and which need to push notifications regarding behaviour occurring to other parts of our apps.

Whilst it may not always be the best solution to every problem, it remains 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 the pub/sub pattern actually stem from its main benefit. 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.

Implementations

One of the benefits of design patterns is that once we understand the basics behind how a particular pattern works, being able to interpret an implementation of it becomes significantly more straightforward. Luckily, popular JavaScript libraries such as dojo and YUI already have utilities that can assist in easily implementing your own pub/sub system.

For those wishing to use the pub/sub pattern with vanilla JavaScript (or another library) AmplifyJS includes a clean, library-agnostic implementation of pub/sub that can be used with any library or toolkit. ScriptJunkie also has a tutorial on how to get started with Amplify's pub/sub that was recently published. You can of course also write your own implementation from scratch or also check out either PubSubJS or OpenAjaxHub, both of which are also library-agnostic.

jQuery developers have quite a few options for pub/sub (in addition to Amplify) and can opt to use one of the many well-developed implementations ranging from Peter Higgins's jQuery

plugin to Ben Alman's (optimized) gist on GitHub. Links to just a few of these can be found below.

Tutorial

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 walk through of a trimmed down version of Morgan Roderick's PubSubJS, which I've put together below. This demonstrates the core concepts of subscribe, 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 original observer pattern.

Sample Pub/Sub implementation

```
var PubSub = {};

(function(p){

   "use strict";
   var topics = {},
         lastUid = -1;


   var publish = function( topic , data){

       if ( !topics.hasOwnProperty( topic ) ){
           return false;
       }


       var notify = function(){
           var subscribers = topics[topic],
               throwException = function(e){
               return function(){
                   throw e;
               };

           };

           for ( var i = 0, j = subscribers.length; i
```

Example 1: Basic use of publishers and subscribers

This could then be easily used as follows:

```
// a sample subscriber (or observer)

var testSubscriber = function( topics , data ){
   console.log( topics + ": " + data );

};



// add the function to the list of subscribers to a particular topic
// maintain the token (subscription instance) to enable unsubscription later

var testSubscription = PubSub.subscribe( 'example1', testSubscriber );



// publish a topic or message asyncronously

PubSub.publish( 'example1', 'hello scriptjunkie!' );

PubSub.publish( 'example1', ['test','a','b','c'] );

PubSub.publish( 'example1', [{'color':'blue'},{'text':'hello'}] );



// unsubscribe from further topics
setTimeout(function(){
   PubSub.unsubscribe( testSubscription );
}, 0);



// test that we've fully unsubscribed
PubSub.publish( 'example1', 'hello again!');
```

Real-time stock market application

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, as well as an underlying data model. When the data model changes, the application will need to update the grid and counter. In this scenario, our subject is the data model and the observers are the grid and counter.

When the observers receive notification that the model itself has changed, they can update themselves accordingly.

Example 2: UI notifications using pub/sub

In the following example, we limit our usage of pub/sub to that of a notification system. Our subscriber is listening to the topic 'dataUpdated' to find out when new stock information is available. It then triggers 'gridUpdate' which goes on to call hypothetical methods that pull in the latest cached data object and re-render our UI components.

Note: the Mediator pattern is occasionally used to provide a level of communication between UI components without requiring that they communicate with eac

h other directly. For example, rather than tightly coupling our applications, we can have widgets/components publish a topic when something interesting happens. A mediator can then subscribe to that topic and call the relevant methods on other components.

```
var grid = {

   refreshData: function(){
       console.log('retrieved latest data from data cache');
       console.log('updated grid component');
   },

   updateCounter: function(){
       console.log('data last updated at: ' + getCurrentTime());
   }

};



//a very basic mediator

var gridUpdate = function(topics, data){
     grid.refreshData();
     grid.updateCounter();
}


var dataSubscription = PubSub.subscribe( 'dataUpdated', gridUpdate );
PubSub.publish( 'dataUpdated', 'new stock data available!' );
PubSub.publish( 'dataUpdated', 'new stock data available!' );


function getCurrentTime(){

   var date = new Date(),
         m = date.getMonth() + 1,
         d = date.getDate(),
         y = date.getFullYear(),
         t = date.toLocaleTimeString().toLowerCase(),
        return (m + '/' + d + '/' + y + ' ' + t);

}
```

Whilst there's nothing terribly wrong with this, there are more optimal ways that we can utilize pub/sub to our advantage.

Example 3: Taking notifications further

Rather than just notifying our subscribers that new data is available, why not actually push the new data through to gridUpdate when we publish a new notification from a publisher. In this next example, our publisher will notify subscribers with the actual data that's been updated as well as a timestamp from the data-source of when the new data was added.

In addition to avoiding data having to be read from a cached store, this also avoids client-side calculation of the current time whenever a new data entry gets published.

```
var grid = {

   addEntry: function(data){

       if (data !== 'undefined') {

          console.log('Entry:'

                      + data.title

                      + ' Changenet / %'

                      + data.changenet

                      + '/' + data.percentage + ' % added');

       }

   },

   updateCounter: function(timestamp){
       console.log('grid last updated at: ' + timestamp);
```

```
   }
};
```

```
var gridUpdate = function(topics, data){
       grid.addEntry(data);
       grid.updateCounter(data.timestamp);
}
```

```
var gridSubscription = PubSub.subscribe( 'dataUpdated', gridUpdate );

PubSub.publish('dataUpdated',   { title: "Microsoft shares", changenet: 4, percentage:
33, timestamp: '17:34:12'  });

PubSub.publish('dataUpdated',   { title: "Dell shares", changenet: 10, percentage: 20,
timestamp: '17:35:16'  });
```

Example 4: 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 pub/sub 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 jQuery.tmpl plugin.

HTML/Templates

```
<script id="userTemplate" type="text/x-jquery-tmpl">
   <li>${user}
</script>


<script id="ratingsTemplate" type="text/x-jquery-tmpl">
   <li><strong>${movie}</strong> was rated ${rating}/5
</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"ected>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>
```

JavaScript

```
(function($) {


  var movieList = [],
     userList  = [];



  /* subscribers */

  $.subscribe( "/new/user", function( e, userName ){

     if(userName.length){
         userList.push({user: userName});
         $( "#userTemplate" ).tmpl( userList[userList.length - 1] ).appendTo( "#users"
);
    }

  });



  $.subscribe( "/new/rating", function( e, movieTitle, userRating ){

     if(movieTitle.length){
       movieList.push({ movie: movieTitle, rating: userRating});
       $( "#ratingsTemplate" ).tmpl( movieList[movieList.length - 1] ).appendTo( "#rati
ngs" );
     }

  });



  $('#add').bind('click', function(){

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

         $.publish('/new/user',  strUser  );
         $.publish('/new/rating',  [ strMovie, strRating] );

    });

})(jQuery);
```

Example 5: 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. This is something Rebecca Murphey touched on in her pub/sub screencast and is another reason why pub/sub is favoured by so many developers in the community.

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. One could simply add all of their 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-behaviour) without rewriting parts of the code multiple times. Rather than having to go back through each layer that calls the same data-source and generalizing them later on, we can use pub/sub from the start and save time.

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

Notice how in our sample below, one topic notification is made when a user in dicates they want 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 su bscribers to then decide how to use knowledge of these events (or the data re turned). The benefits of this are that, if we wanted, we could have 10 differ ent subscribers utilizing the data returned in different ways but as far as t he 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 co ncerns can make the overall design of your code a little cleaner.

HTML/Templates:

```
<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/x-jquery-tmpl">
    {{each(i, items) items}}
            <li><p><img src="${items.media.m}"/></p></li>
    {{/each}}
</script>
```

JAVASCRIPT:

```
(function($) {

   $('#flickrSearch').submit(function( e ){

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

       if(!tags){return;}
       $.publish('/search/tags', [ $.trim(tags) ]);

   });



   $.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 ]);
       });

   });



   $.subscribe('/search/tags', function(tags){
       $('#searchResults').html('

Searched for:' + tags + '

');
   });


   $.subscribe('/search/resultSet', function(results){

       var holder = $('#searchResults');
       holder.html();
       $('#resultTemplate').tmpl(results).appendTo(holder);

   });


});
```

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 pre-written implementations mentioned today and just giving it a try out. It's one of the easier design patterns to get started with but also one of the most powerful.

MVC And MVP

In this section, we're going to review two very important architectural patterns - MVC (Model-View-Controller) and MVP (Model-View-Presenter). In the past both of these patterns have been heavily used for structuring desktop and server-side applications, but it's only been in recent years that come to being applied to JavaScript.

As the majority of JavaScript developers currently using these patterns opt to utilize libraries such as Backbone.js for implementing an MVC/MV*-like structure, we will compare how modern solutions such as it differ in their interpretation of MVC compared to classical takes on these patterns.

Let us first now 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, user-input and coordinating both t

he models and views. 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 “Design Patterns: Elements of Reusable Object-Oriented Software” (The "GoF" book) in 1994, 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's mutated quite heavily since the days of it's origin. Back in the 70's, graphical user-interfaces were far and few between and a concept known as Separated Presentation began to be used as a means to make a clear division between domain objects which modelled concepts in the real world (e.g a photo, a person) and the presentation objects which were rendered to the user's screen.

The Smalltalk-80 implementaion 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 Domain element was known as a Model and were ignorant of the user-interface (Views and Controllers)
- Presentation was taken care of by the View and the Controller, but there wasn't just a single view and controller. A View-Controller pair was required for each element being displayed on the screen and so there was no true separation between them
- The Controller's role in this pair was handling user input (such as key-presses and click events), doing something sensible with them.
- The Observer pattern was relied upon for updating the View whenever the Model changed

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as a Publish/Subscribe system) was included as a part of MVC's architecture many decades ago. In Smalltalk-80's MVC, the View and Controller both observe 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 70's, 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. You've likely come across at least one of these such frameworks, but they include the likes of Backbone, Ember.js and JavaScriptMVC. Given the importance of avoiding "spaghetti" code, a term which 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.

We know that MVC is composed of three core components:

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 meta-data. 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 also have multiple views observing it. If say, our photo model contained meta-data such as its location (longitude and latitude), friends that were present in the a 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 below.

```
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() );
    }

});
```

Should you read any of the older texts on MVC, you may come across a description of models as also managing application 'state'. In JavaScript applications "state" has a different meaning, typically referring to the current "state" i.e view or sub-view (with specific data) on a users screen at a fixed point. State is a topic which 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. 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 meta-data.

The actual task of updating the model falls to controllers (which we'll 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 it's subscribers so that through the Observer pattern we can trigger the view to update when the model changes.

You 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 behaviour 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 it's 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 inperformantly 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 your standard markup, it can quickly become both difficult to read and more importantly, maintain such disasters, especially when building non-trivially 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 deliminated using a variable syntax (e.g {{name}}) and frameworks are typically smart enough to accept data in a JSON form (of 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.

Below we can see two examples of HTML templates. One implemented using the popular Handlebars.js framework and another using Underscore's templates.

Handlebars.js:

```
<li class="photo">
  <h2>{{caption}}</h2>
```

```
  <img class="source" src="{{src}}"/>
  <div class="meta-data">
    {{metadata}}
  </div>
</li>
```

Underscore.js Microtemplates:

```
<li class="photo">
  <h2><%= caption %></h2>
  <img class="source" src="<%= src %>"/>
  <div class="meta-data">
    <%= metadata %>
  </div>
</li>
```

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 an intermediary between models and views which are classically responsible for two tasks: they both update the view when the model changes and update 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.

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 re-interpret the C in MVC to mean

something they feel makes more sense. The issue with this however is that it is subjective, increases the complexity in both understanding the classical MVC pattern and of course 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 which 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 you be relying on advice from classical literature on the former to help with the latter.

Controllers in another library (Spine.js) vs Backbone.js

Spine.js

We now know that controllers are traditionally responsible for updating the view when the model changes (and similarly the model when the user updates the view). As the framework we'll be discussing in this book (Backbone) doesn't have it's **own** explicit controllers, it can be useful for us to review the controller from another MVC framework to appreciate the difference in implementations. 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 editd the photo meta-data) the corresonding model does too.

Note: We won't be delving heavily into Spine.js at all, but will just take a ten-foot view of what it's 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 meta-data. Similarly, if the photo gets deleted from the gallery, we remove it from the view. In case you were wondering about the `tmpl()` function in the code snippet: in the `render()` function, we're using this to render a JavaScript template called #photoTemplate which simply returns a HTML string used to replace the controller's current element.

What this provides us with is a very lightweight, simple way to manage changes between the model and the view.

Backbone.js

Later on 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 the Backbone.View and Backbone.Router. Some time ago Backbone did once come with it's 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 your 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 });

    something.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 you 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

Delving deeper

Right now, you likely have a basic understanding of what the MVC pattern provides, but for the curious, we can explore it a little further.

The GoF (Gang of Four) 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 other classical design patterns: the Observer (Pub/Sub), Strategy and Composite patterns. Depending on how

MVC has been implemented in a framework, it may also use the Factory and Decorator patterns.

As we've discussed, models represent application data whilst views are what the user is presented on screen. As such, MVC relies on Pub/Sub for some of its core communication (something that surprisingly isn't cover in many articles about the MVC pattern). When a model is changed it notifies the rest of the application it has been updated. The controller then updates the view accordingly. The observer nature of this relationship is 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 implement), one of the goal's 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 shares 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 you grok how the framework expects to be used.

MVP

Model-view-presenter (MVP) is a derivative of the MVC design pattern which 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. Whilst 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 & Presenters

The P in MVP stands for presenter. It's a component which 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.

The most common implementation of MVP is one which uses a Passive View (a view which is for all intents and purposes "dumb"), containing little to no logic. MVP models are almost identical to MVC models and handle application data. The presenter acts as a mediator which talks to both the view and model, however both of these are isolated from each other. They effectively bind models to views, a responsibility which was previously held by controllers in MVC. Presenters are at the heart of the MVP pattern and as you can guess, incorporate the presentation logic behind views.

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 which presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of your 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 which 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) plugins (such as Derick Bailey's Backbone.ModelBinding plugin) 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 more easy 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 you are implementing MVP in (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 you may have with MVC will likely hold true for MVP given that the differences between them are mainly semantic. As long as you are cleanly separating concerns into models, views and controllers (or presenters) you should be achieving most of the same benefits regardless of the pattern you opt for.

MVC, MVP and Backbone.js

There are very few, if any architectural JavaScript frameworks that claim to implement the MVC or MVC 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 your application and yet still consider it a flavor of MVC.

Backbone contributor Irene Ros (of Boston-based Bocoup) 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 however 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 the 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() {
      _.bindAll(this, 'render');
```

```
      this.model.bind('change', this.render);
      this.model.bind('destroy', this.remove);
    },

    // 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 user 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 which are organized, clean and can be easily maintained.

Decorator Pattern

In this section we're going to continue exploring the **decorator** - a structural designpattern that promotes code reuse and is a flexible alternative to subclassing. This pattern is also useful for modifying existing systems where you may wish to add additional features to objects without the need to change the underlying code that uses them.

Traditionally, the decorator is defined as a design pattern that allows behaviour to be added to

an existing object dynamically. The idea is that the decoration itself isn't essential to the base functionality of an object otherwise it would be baked into the 'superclass' object itself.

Subclassing

For developers unfamiliar with subclassing, here is a beginner's primer on them before we dive further into decorators: subclassing is a term that refers to inheriting properties for a new object from a base or 'superclass' object.

In traditional OOP, 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 it's own methods, including those that override methods originally defined by A.

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

In order 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 subclassExample = subclassExample || {};
subclassExample = {
    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's imagine we want to add distinct properties to distinguish a Person from a Superhero whilst inheriting the properties of the Person 'superclass'. As superheroes share many common traits with normal people (eg. 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 subclassExample.Person( "Clark" , "Kent" );

//Define a subclass constructor for for 'Superhero':
subclassExample.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.
    */
    subclassExample.Person.call(this, firstName, lastName);
    //Finally, store their powers, a new array of traits not found in a norma
l 'Person'
    this.powers = powers;
}
subclassExample.Superhero.prototype = new subclassExample.Person;
var superman = new subclassExample.Superhero( "Clark" ,"Kent" , ['flight','he
at-vision'] );
console.log(superman); /* includes superhero props as well as gender*/
```

The Superhero definition creates an object which descends from Person. Objects of this type have properties of the objects that are above it 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 it's object.

So where do decorators come in?

Decorators

As we've previously covered, Decorators are used when it's necessary to delegate responsibilities to an object where it doesn't make sense to subclass it. A common reason for this is that the number of features required demand for a very large quantity of subclasses. Can you imagine having to define hundreds or thousands of subclasses for a project? It would likely become unmanagable fairly quickly.

To give you a visual example of where this is an issue, imagine needing to define new kinds of Superhero: SuperheroThatCanFly, SuperheroThatCanRunQuickly and SuperheroWithXRayVision.

Now, what if s superhero had more than one of these properties?. We'd need to define a subclass called SuperheroThatCanFlyAndRunQuickly , SuperheroThatCanFlyRunQuicklyAndHasXRayVision etc - effectively, one for each possible combination. As you can see, this isn't very manageable when you factor in 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 using inheritance, where we're used

to extending objects linearly, we work with a single base object and progressively add decorator objects which provide the additional capabilities. The idea is that rather than subclassing, we add (decorate) properties or methods to a base object so its a little more streamlined.

The extension of objects is something already built into JavaScript and as we know, objects can be extended rather easily with properties being included at any point. With this in mind, a very very simplistic decorator may be implemented as follows:

Example 1: Basic decoration of existing object constructors with new functionality

```
function vehicle( vehicleType ){
    /*properties and 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);
/*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);
/*vehicle:truck, model:CAT, color: blue*/
/*Demonstrate 'vehicle' is still unaltered*/
var secondInstance = new vehicle('car');
console.log(secondInstance);
/*as before, vehicle: car, model:default, license: 00000-000*/
```

This type of simplistic implementation is something you're likely familiar with, but it doesn't really demonstrate some of the other strengths of the pattern. 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 modelled around a Macbook purchase.

We're then going to look at psuedo-classical decorators.

Example 2: Simply decorate objects with multiple decorators

```
//What we're going to decorate
function MacBook() {
    this.cost = function () { return 997; };
    this.screenSize = function () { return 13.3; };
}

/*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);
console.log(mb.cost()); //1522
console.log(mb.screenSize()); //13.3
```

Here, the decorators are overrriding the superclass .cost() method to return the current price of the Macbook plus with the cost of the upgrade being specified. It's considered a decoration as

the original Macbook object's constructor methods which are not overridden (eg. screenSize()) as well as any other properties which we may define as a part of the Macbook remain unchanged and in tact.

As you can probably tell, there isn't really a defined 'interface' in the above example and duck typing is used to shift the responsibility of ensuring an object meets an interface when moving from the creator to the receiver.

Pseudo-classical decorators

We're now going to examine the variation of the decorator presented 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'll define in more detail shortly.

Note: This particular variation of the decorator pattern is provided for reference purposes. If you find it overly complex for your application's needs, I recommend sticking to one the simplier implementations covered earlier, but I would still read the section. If you haven't yet grasped how decorators are different from subclassing, it may help!.

Interfaces

PJDP describes the decorator as a pattern 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.

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

So, why would you 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 classes 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 constructor/object based on the methods it implements.

```
var TodoList = new Interface('Composite', ['add', 'remove']);
var TodoItem = new Interface('TodoItem', ['save']);
```

```
// TodoList class
var myTodoList = function(id, method, action) {
        // implements TodoList, TodoItem
...
};
...
function addTodo( todoInstance ) {
        Interface.ensureImplements(todoInstance, TodoList, TodoItem);
        // This function will throw an error if a required method is not impl
emented,
        // halting execution of the function.
        //...
}
```

where Interface.ensureImplements provides strict checking. If you would like to explore interfaces further, I recommend looking at Chapter 2 of Pro JavaScript design patterns. For the Interface class used above, see here.

The biggest problem with interfaces is that, as there isn't built-in support for them in JavaScript, there's a danger of us attempting to emulate the functionality of another language, however, we're going to continue demonstrating their use just to give you a complete view of how the decorator is implemented by other developers.

This variation of decorators and 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 you to 'decorate' your macbook with a number of enhancements for an additional fee.

Enhancements can include upgrades to 4GB or 8GB 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'd 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 astray use the same interface.

Here's the interface we're going to define for the Macbook:

```
var Macbook = new Interface('Macbook', ['addEngraving', 'addParallels', 'add4
GBRam', '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(){
                return 900.00; //base price.
        }
};
```

We're not going to worry about the actual implementation at this point as we'll shortly be passing on all method calls that are made on them.

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 is just calling 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

overriden as per 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 you can see, most of this is relatively easy 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've learned.

```
//Instantiation of the macbook
var myMacbookPro = new MacbookPro();
//This will return 900.00
console.log(myMacbookPro.getPrice());
//Decorate the macbook
myMacbookPro = new CaseDecorator( myMacbookPro ); /*note*/
//This will return 945.00
console.log(myMacbookPro.getPrice());
```

An important note from PJDP is that in the line denoted *note*, Harmes and Diaz claim that it's important not to create a separate variable to store the instance of your decorators, opting for the same variable instead. The downside to this is that we're unable to access the original macbook object in our example, however we technically shouldn't need to further.

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. This makes maintaining applications of this type significantly more straight-forward.

Implementing decorators with jQuery

As with other patterns I"ve covered, there are also examples of the decorator pattern that can be implemented with jQuery. jQuery.extend() allows you to extend (or merge) two or more objects (and their properties) together into a single object either at run-time 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 'options', which we'll make available through 'settings'. We must:

(a) Leave 'defaults' in an untouched state where we don't lose the ability to access the properties or functions found in it a later point (b) 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');
             }
            },
    settings:{},
    printObj: function(obj) {
            var arr = [];
            $.each(obj, function(key, val) {
            var next = key + ": ";
```

```
            next += $.isPlainObject(val) ? printObj(val) : val;
            arr.push( next );
      });
      return "{ " +  arr.join(", ") + " }";
    }

}
/* merge defaults and options, without modifying defaults */
decoratorApp.settings = $.extend({}, decoratorApp.defaults,decoratorApp.optio
ns);
/* what we've done here is decorated defaults in a way that provides access t
o the properties and functionality it has to offer (as well as that of the de
corator 'options'). defaults itself is left unchanged*/
$('#log').append("<div><b>settings -- </b>" + decoratorApp.printObj(decorator
App.settings) + "</div><div><b>options -- </b>" + decoratorApp. printObj(deco
ratorApp.options) + "</div><div><b>defaults -- </b>" +decoratorApp.printObj(d
ecoratorApp.defaults) + "</div>" );
/*
settings -- { validate: true, limit: 5, name: bar, welcome: function (){ cons
ole.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 (){ con
sole.log('welcome!'); } }
*/
```

Pros and cons of the pattern

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 avoids us needing to rely on large numbers of subclasses to get the same benefits.

There are however drawbacks that you should be aware of when implementing the pattern. If poorly managed, it can significantly complicate your application's architecture as it introduces many small, but similar objects into your 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 you keep a handle on how widespread you use the decorator in your application you should be fine on both counts.

Namespacing Patterns

In this section, I'll be discussing both intermediate and advanced patterns for namespacing in JavaScript. We're going to begin with the latter, however if you're new to namespacing with the language and would like to learn more about some of the fundamentals, please feel free to skip to the section titled 'namespacing fundamentals' to continue reading.

What is namespacing?

In many programming languages, namespacing is a technique employed to avoid **collisions** with other objects or variables in the global namespace. They're also extremely useful for helping organize blocks of functionality in your application into easily manageable groups that can be uniquely identified.

In JavaScript, namespacing at an enterprise level is critical as it's important to safeguard your code from breaking in the event of another script on the page using the **same** variable or method names as you 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 you do your best to similarly not prevent other developer's scripts executing due to the same issues.

Whilst 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.

Advanced namespacing patterns

In this section, I'll be exploring some advanced patterns and utility techniques that have helped me when working on larger projects requiring a re-think of how application namespacing is approached. I should state that I'm not advocating any of these as *the* way to do things, but rather just ways that I've found work in practice.

Automating nested namespacing

As you're probably aware, a nested namespace provides an organized hierarchy of structures in an application and an example of such a namespace could be the following: *application.utilities.drawing.canvas.2d*. In JavaScript the equivalent of this definition using the object literal pattern would be:

```
var application = {
            utilities:{
                    drawing:{
```

```
                        canvas:{
                                2d:{
                                        /*...*/
                                }
                        }
                }
        }
};
```

Wow, that's ugly.

One of the obvious challenges with this pattern is that each additional depth you wish to create requires yet another object to be defined as a child of some parent in your top-level namespace. This can become particularly laborious when multiple depths are required as your 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 using a convenience method that takes a single string argument for a nest, parses this and automatically populates your 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 re-use 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, i;

    if (parts[0] == "myApp") {
        parts = parts.slice(1);
    }

    pl = parts.length;
    for (i = 0; i < pl; i++) {
        // create a property if it doesnt exist
        if (typeof parent[parts[i]] == 'undefined') {
```

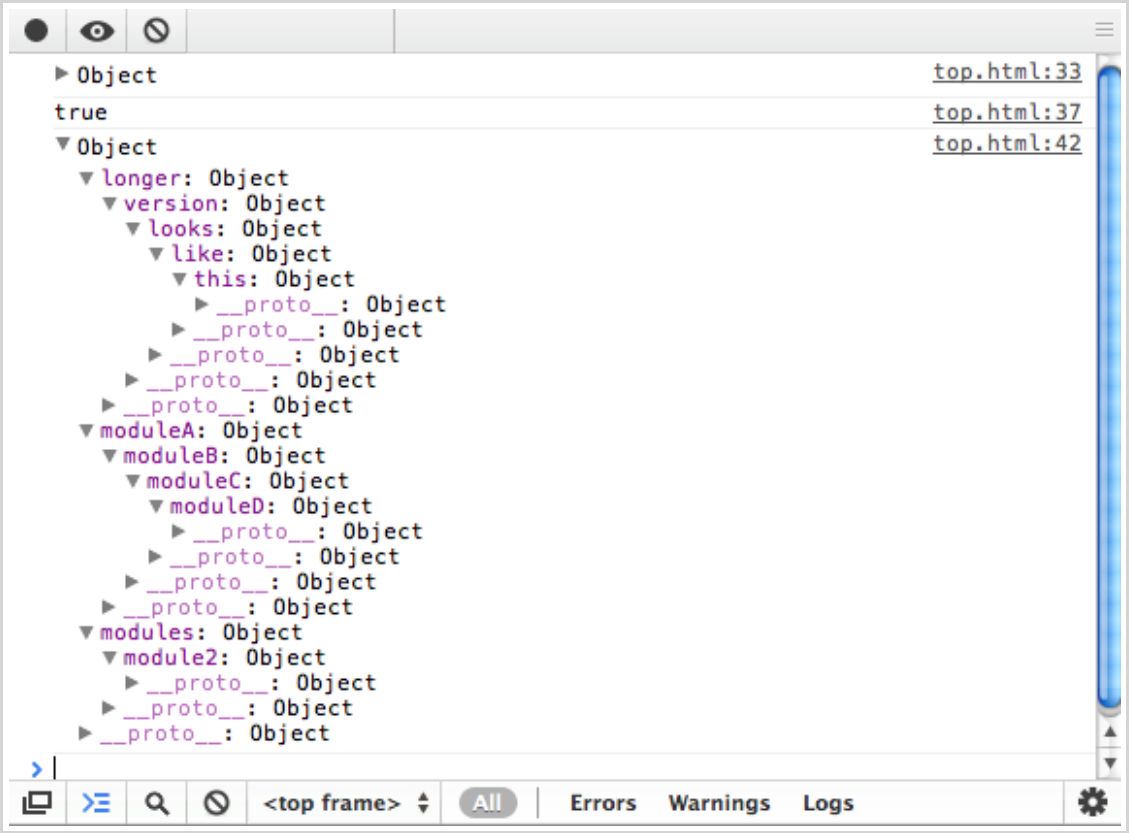
```
            parent[parts[i]] = {};
        }

        parent = parent[parts[i]];
    }

    return parent;
}

// sample 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
console.log(mod == myApp.modules.module2); //true
// 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);
```

Web inspector output:



Note how 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. This works exceedingly well when defining purely namespaces alone, but can seem a little less flexible when you want to define both functions and properties at the same time as declaring your namespaces. Regardless, it is still incredibly powerful and I regularly use a similar approach in some of my projects.

Dependency declaration pattern

In this section we're going to take a look at a minor augmentation to the nested namespacing pattern you may be used to seeing in some applications. We all know that local references to objects can decrease overall lookup times, but let's apply this to namespacing 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 (eg.myApp). It's also both more convenient and more performant than accessing nested properties/sub-namespaces 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 your function scope (using the single-variable pattern) and calls this a dependancy declaration pattern. One if the benefits this offers is a decrease in locating dependencies and resolving them, should you have an extendable architecture that dynamically loads modules into your 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 (eg. see jQuery's $.extend), however, if you're looking to extend object (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
// (eg. '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();
```

If you do happen to be using jQuery in your application, you can achieve the exact same object namespact extensibility using $.extend as seen below:

```
// 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.

Namespacing Fundamentals

Namespaces can be found in almost any serious JavaScript application. Unless you're working with a code-snippet, it's imperative that you do your best to ensure that you're implementing namespacing correctly as it's not just simple to pick-up, it'll also avoid third party code clobbering your own. The patterns we'll be examining in this section are:

1. Single global variables
2. Object literal notation
3. Nested namespacing
4. Immediately-invoked Function Expressions
5. Namespace injection

1.Single global variables

One popular pattern for namespacing in JavaScript is opting for a single global variable as your 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 you have in the page.

One solution to this problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is you select a unique prefix namespace you 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 = {};
funcion myApplication_myMethod(){ /*..*/ }
```

This is effective from the perspective of trying to lower 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 your application starts to grow. There is also quite a heavy reliance on your 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 here.

2. Object literal notation

Object literal notation can be thought of as an object containing a collection of key:value pairs with a colon separating each pair of keys and values. It's syntax requires a comma to be used after each key:value pair with the exception of the last item in your object, similar to a normal array.

```
var myApplication = {
    getInfo:function(){ /**/ },
```

```
    // we can also populate our object literal to support
    // further object literal namespaces containing anything
    // really:
    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're 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 code at the very top of the next sample demonstrates the different ways in which you can check to see if a variable (object namespace) already exists before defining it. You'll commonly see developers using Option 1, however Options 3 and 5 may be considered more thorough and Option 4 is considered a good best-practice.

```
// This doesn't check for existence of 'myApplication' in
// the global namespace. Bad practice as you 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: var myApplication = myApplication = myApplication || {}
Option 4: myApplication || (myApplication = {});
Option 5: var myApplication = myApplication === undefined ? {} : myApplicatio
n;

*/
```

There is of course a huge amount 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 this next pattern when returning an interface for other developers to use. It's a variation on the module pattern where the core structure of the pattern is an IIFE, however the returned interface is an object literal:

```
var namespace = (function () {

    // defined within the local scope
    var privateMethod1 = function () { /* ... */ }
    var privateMethod2 = function () { /* ... */ }
    var privateProperty1 = 'foobar';

    return {
        // the object literal returned here can have as many
        // nested depths as you 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 object literals 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 your code.

A possible downside however is that object literals have the potential to grow into long syntactic constructs. Opting to take advantage of the nested namespace pattern (which also uses the same pattern as it's base)

This pattern has a number of other useful applications too. In addition to namespacing, it's often of benefit to decouple the default configuration for your application into a single area that can be easily modified without the need to search through your entire codebase just to alter them - object literals work great for this purpose. Here's an example of a hypothetical object literal for configuration:

```
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 there are really only minor syntactical differences between the object literal pattern and a standard JSON data set. If for any reason you wish to use JSON for storing your configurations instead (e.g. for simpler storage when sending to the back-end), feel free to. For more on the object literal pattern, I recommend reading Rebecca Murphey's excellent article on the topic.

3. 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');
```

Yahoo's YUI framework uses the nested object namespacing pattern regularly and at AOL we also use this pattern in many of our main 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();
```

You 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 your application in a similar fashion to what you may be used to in other languages. The only real caveat however is that it requires your browser's JavaScript engine first locating the myApp object and then digging down until it gets to the function you 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 vs the 'nested' approach to be quite negligible.

4. Immediately-invoked Function Expressions (IIFE)s

An IIFE is effectively an unnamed function which is immediately invoked after it's been defined. 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.

This is one of the many reasons why IIFEs are a popular approach to encapsulating application logic to protect it from the global namespace. You've probably come across this pattern before under the name of a self-executing (or self-invoked) anonymous function, however I personally prefer Ben Alman's naming convection for this particular pattern as I believe it to be both more descriptive and more accurate.

The simplest version of an IIFE could be the following:

```
// 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(); }
```

whilst 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
console.log(namespace.foobar); // foobar
namescpace.sayHello(); // hello world

// assigning new properties
namespace.foobar2 = "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 || {});
```

namespace.sayGoodbye(); //goodbye

That's it for IIFEs for the time-being. If you would like to find out more about this pattern, I recommend reading both Ben's IIFE post and Elijah Manor's post on namespace patterns from C#.

5. Namespace injection

Namespace injection is another variation on the IIFE where 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 behaviour to multiple objects or namespaces and can come in useful when applying a set of base methods to be built on later (eg. getters and setters).

The disadvantages of this pattern are that there may be easier or more optimal approaches to achieving this goal (eg. deep object extension / merging) which I cover earlier in the article..

Below we can see an example of this pattern in action, where we use it to populate the behaviour 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'

// testing
console.log(myApp); //the now populated namespace
console.log(myApp.utils.getValue()); // test get
myApp.utils.setValue(25); // test set
console.log(myApp.utils.getValue());
console.log(myApp.utils.tools.diagnose());
```

Angus Croll has also previously suggested the idea of using the call API to provide a natural separation between contexts and arguments. This pattern can feel a lot more like a module creator, but as modules still offer an encapsulation solution, I'll briefly cover it for the sake of thoroghness:

```
// 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, but I'd really only suggest using it where explicitly declaring your functionality within an object/closure for direct access doesn't make sense.

Reviewing the namespace patterns above, the option that I would personally use for most larger applications is nested object namespacing with the object literal pattern.

IIFEs and single global variables may work fine for applications in the small to medium range, however, larger codebases requiring both namespaces and deep sub-namespaces 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 you time in the long-run.

Flyweight

The Flyweight pattern is considered a useful classical solution for code that's repetitive, slow and inefficient - for example: situations where we might create a large number of similar

objects.

It's of particular use in JavaScript where code that's complex in nature may easily use all of the available memory, causing a number of performance issues. Interestingly, it's been quite underused in recent years. Given how reliant we are on JavaScript for the applications of today, both performance and scalability are often paramount and this pattern (when applied correctly) can assist with improving both.

To give you some quick historical context, the pattern is named after the boxing weight class that includes fighters weighing less than 112lb - Poncho Villa being the most famous fighter in this division. It derives from this weight classification as it refers to the small amount of weight (memory) used.

Flyweights are an approach to taking several similar objects and placing that shared information into a single external object or structure. The general idea is that (in theory) this reduces the resources required to run an overall application. The flyweight is also a structural pattern, meaning that it aims to assist with both the structure of your objects and the relationships between them.

So, how do we apply it to JavaScript?

There are two ways in which the Flyweight pattern can be applied. The first is on the data-layer, where we deal with the concept of large quantities of similar objects stored in memory. The second is on 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 you wish to have some similar behaviour.

As the data-layer is where the flyweight pattern is most used traditionally, we'll take a look at this first.

Flyweight and the data layer

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 your objects which they absolutely can't 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, meaning we're able to reduce the overall quantity of objects down 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, however as Dustin Diaz correctly points out in Pro JavaScript Design patterns, one approach to this to have the manager object contain a central database of the extrinsic states and the flyweight objects which they belong to.

Converting code to use the Flyweight pattern

Let's now demonstrate some of these concepts using the idea of a system to manage all of the books in a library. The important meta-data 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 they've 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:

```
var Book = function( id, title, author, genre, pageCount,publisherID, ISBN, c
heckoutDate, 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;
   },
/*other getters not shown for brevity*/
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, you'll 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 whilst 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 meta-data 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 you 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's data is now segmented, a factory can be used for instantiation.

A Basic Factory

Let's now define a very basic factory. What we're going to have it do is 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 = {};
   return{
       createBook: function(title, author, genre,pageCount,publisherID,ISBN){
       /*Find out if a particular book meta-data combination has been created
 before*/
           var existingBook = existingBooks[ISBN];
           if(existingBook){
                   return existingBook;
               }else{
               /*if not, let's create a new instance of it 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 them 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,p
ublisherID,ISBN);
            bookRecordDatabase[id] ={
               checkoutMember: checkoutMember,
               checkoutDate: checkoutDate,
               dueReturnDate: dueReturnDate,
               availability: availability,
               book: book;

           };
       },
    updateCheckoutStatus: function(bookID, newStatus, checkoutDate, checkoutM
ember,     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].d
ueReturnDate);
    }
  };
});
```

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) which is considerable 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 more memory.

The Flyweight pattern and the DOM

In JavaScript, functions are effectively object descriptors and all functions are also JavaScript objects internally. The goal of the pattern here is thus to make triggering objects have little to no responsibility for the actions they perform and to instead abstract this responsibility up to a global manager. One of the best metaphors for describing the pattern 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 raise to the surface (the bubbling) a fly sitting on the top flies away when the bubble reaches the surface (the action). In this example you 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'.

As jQuery is accepted as one of the best options for DOM-manipulation and selection, we'll be using it for our DOM-related examples.

Example 1: Centralized event handling

For our first pratical example, consider scenarios where you may have a number of similar

elements or structures on a page that share similar behaviour when a user-action is performed against them.

In JavaScript, there's a known bubbling effect in the language so that if an element such as a link or button is clicked, that event is bubbled up to the parent, informing them that something lower down the tree has been clicked. We can use this effect to our advantage.

Normally what you might do when constructing your own accordion component, menu or other list-based widget is bind a click event to each link element in the parent container. 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 as simple or as complex logic needed.

The benefit here is that we're converting many independent objects into a few shared ones (potentially saving on memory), similar to what we were doing with our first JavaScript example.

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 behaviour 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 encapsulate our flyweight logic whilst jQuery is used to bind the initial click to a container div. In order 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.

HTML

```
<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&am
```

```
p;address=london%2C%20england&amp;width=500...gt;"</iframe>
        </span>
    </div>
</div>
```

JAVASCRIPT

```
stateManager = {
    fly: function(){
        var self =  this;
        $('#container').unbind().bind("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');
    }
});
```

Example 2: Using the Flyweight for Performance Gains

In our second example, we'll reference some useful performance gains you can get from applying the flyweight pattern to jQuery.

James Padolsey previously wrote a post called '76 bytes for faster jQuery' where he reminds us of an important point: every time jQuery fires off a callback, regardless of type (filter, each, event handler), you'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 constructed every time.

Rather than doing this:

```
$('div').bind('click', function(){
 console.log('You clicked: ' + $(this).attr('id'));
});
you should avoid using the DOM element to create a jQuery object (with the ov
erhead that comes with it) and just use the DOM element itself like this:
```

```
$('div').bind('click', function(){
 console.log('You clicked: ' + this.id);
});
```

Now with respect to redundant wrapping, where possible with jQuery's utility methods, it's better to use jQuery.N as opposed to jQuery.fn.N where N represents a utility such as each. Because not all of jQuery's methods have corresponding single-node functions, Padolsey devised the idea of jQuery.single.

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 is a flyweight as we're consolidating multiple possible objects into a more central singular structure.

```
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').bind('click', function(){
   var html = jQuery.single(this).next().html();
   console.log(html);
 });
```

Note that although we may believe that simply caching our jQuery code may offer just as 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.

Modules

In this section we're going to continue our exploration of the Module pattern and the broader concept of a 'module'.

Modules are an integral piece of any robust application's architecture and typically help in keeping the code for a project organized. In JavaScript, there are several options for implementing modules including both the well-known module pattern as well as object literal notation.

Object Literals

The module pattern is based in part on object literals and so it makes sense to review them first. In object literal notation, an object is described as a set of comma-separated name/value pairs enclosured 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.

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. Below you can see an example of a module defined using object literal syntax.

New members may be added to the object using assignment as follows `myModule.property = 'someValue';`

```
var myModule = {
    myProperty : 'someValue',
    // object literals can contain properties and methods.
    // here, another object is defined for configuration
    // purposes:
    myConfig:{
        useCaching:true,
        language: 'en'
    },
    // a very basic method
    myMethod: function(){
        console.log('I can haz functionality?');
    },
    // output a value based on current configuration
    myMethod2: function(){
        console.log('Caching is:' + (this.myConfig.useCaching)?'enabled':'dis
abled');
```

```
    },
    // override the current configuration
    myMethod3: function(newConfig){
        if(typeof newConfig == 'object'){
           this.myConfig = newConfig;
           console.log(this.myConfig.language);
        }
    }
};

myModule.myMethod(); //I can haz functionality
myModule.myMethod2(); //outputs enabled
myModule.myMethod3({language:'fr',useCaching:false}); //fr
```

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

That said, if you're opting for this technique, you 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

As we reviewed earlier in the book, 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 whilst only exposing an interface you wish other parts of your application to use. The pattern is quite similar to an immediately-invoked functional expression (IIFE) except that an object is returned rather than a function.

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 and re-introduced by Eric Miraglia on the YUI blog.

Below you can see an example of 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 and so other parts of your application are unable to directly

read it. It only exists with the module's closure and so the only methods able to access it are those with access to its scope (ie. `addItem()`, `getItem()` etc).

```
var basketModule = (function() {
    var basket = []; //private
    function doSomethingPrivate(){
      //...
    }

    function doSomethingElsePrivate(){
      //...
    }
    return { //exposed to public
        addItem: function(values) {
            basket.push(values);
        },
        getItemCount: function() {
            return basket.length;
        },
        doSomething: doSomethingPrivate(),
        getTotal: function(){
           var q = this.getItemCount(),p=0;
            while(q--){
                p+= basket[q].price;
            }
            return p;
        }
    }
}());
```

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

```
//basketModule is an object with properties which can also be methods
basketModule.addItem({item:'bread',price:0.5});
basketModule.addItem({item:'butter',price:0.3});

console.log(basketModule.getItemCount());
console.log(basketModule.getTotal());

//however, the following will not work:
```

```
console.log(basketModule.basket);// (undefined as not inside the returned obj
ect)
console.log(basket); //(only exists within the scope of the closure)
```

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 which 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 attemping to discover what function(s) threw an exception.
- 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 in order to provide a code-path in their module specific to IE, but we can easily opt for feature detection these days to achieve a similar goal.

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.

How about the module pattern implemented in specific toolkits or frameworks?

Dojo

Dojo provides a convenience method for working with objects called `dojo.setObject()`. This takes as it's 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()
  customStore.setObject("basket.core", (function() {
      var basket = [];
      function privateMethod() {
          console.log(basket);
      }
      return {
          publicMethod: function(){
                  privateMethod();
          }
      };
  }()), store);

});
```

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 do demonstrate how to correctly use the module pattern with the framework.

Below we can see an example of how to define a namespace which 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;
    var privVar1 = 11;

    // private functions
    var btn1Handler = function(button, event) {
        alert('privVar1=' + privVar1);
        alert('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
                });
            }
        }
    };
}(); // end of app
```

YUI

Similarly, we can also implement the module pattern when building applications using YUI. 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:

```
YAHOO.store.basket = function () {

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

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

    return {
        myPublicProperty: "I'm a public property.",
        myPublicMethod: function () {
            YAHOO.log("I'm a public method.");

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

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

}();
```

jQuery

There are a number of ways in which jQuery code unspecific to plugins can be wrapped inside the module pattern. Ben Cherry previously suggested an implementation where 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 which declares a new library and automatically binds up the `init` function to `document.ready` when new libraries (ie. modules) are created.

```
function library(module) {
  $(function() {
```

```
    if (module.init) {
      module.init();
    }
  });
  return module;
}

var myLibrary = library(function() {
   return {
     init: function() {
       /*implementation*/
     }
   };
}());
```

For further reading on the module pattern, see Ben Cherry's article on it here.

Examples Of Design Patterns in jQuery

Now that we've taken a look at vanilla-JavaScript implementations of popular design patterns, let's switch gears and find out what of these design patterns might look like when implemented using jQuery. jQuery (as you may know) is currently the most popular JavaScript library and provides a layer of 'sugar' on top of regular JavaScript with a syntax that can be easier to understand at a glance.

Before we dive into this section, it's important to remember that many vanilla-JavaScript design patterns can be intermixed with jQuery when used correctly because jQuery is still essentially JavaScript itself.

jQuery is an interesting topic to discuss in the realm of patterns because the library actually uses a number of design patterns itself. What impresses me is just how cleanly all of the patterns it uses have been implemented so that they exist in harmony.

Let's take a look at what some of these patterns are and how they are used.

Module Pattern

We have already explored the module pattern previously, but in case you've skipped ahead: the **Module Pattern** allows us to encapsulate logic for a unit of code such that we can have both private and public methods and variables. This can be applied to writing jQuery plugins too, where a private API holds any code we don't wish to expose and a public API contains anything

a user will be allowed to interact with. See below for an example:

```
!function(exports, $, undefined){

    var Plugin = function(){

        // Our private API
        var priv = {},

            // Our public API
            Plugin = {},

            // Plugin defaults
            defaults = {};

        // Private options and methods
        priv.options = {};
        priv.method1 = function(){};
        priv.method2 = function(){};

        // Public methods
        Plugin.method1 = function(){...};
        Plugin.method2 = function(){...};

        // Public initialization
        Plugin.init = function(options) {
            $.extend(priv.options, defaults, options);
            priv.method1();
            return Plugin;
        }

        // Return the Public API (Plugin) we want
        // to expose
        return Plugin;
    }


    exports.Plugin = Plugin;

}(this, jQuery);
```

This can then be used as follows:

```
var myPlugin = new Plugin;
myPlugin.init(/* custom options */);
myPlugin.method1();
```

Lazy Initialization

Lazy Initialization is a design pattern wish allows us to delay expensive processes (eg. the creation of objects) until the first instance they are needed. An example of this is the `.ready()` function in jQuery that only executes a function 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 );
    });
});
```

Whilst it isn't directly used in jQuery core, some developers will be familiar with the concept of LazyLoading via plugins 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 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. Implementing this pattern allows you to treat both individual objects and compositions in a uniform manner. In jQuery, when we're accessing or performing actions on a single DOM element or a collection of elements, we can treat both sets in a uniform manner. This is demonstrated by the code sample below:

```
// Single elements
$('#singleItem').addClass('active');
$('#container').addClass('active');

// Collections of elements
$('div').addClass('active');
$('.item').addClass('active');
$('input').addClass('active');
```

The Wrapper Pattern

The Wrapper Pattern is a pattern which translates an *interface* for a class into a an interface compatible with a specific system. Wrappers basically allow classes to function together which normally couldn't due to their incompatible interfaces. The wrapper 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 a wrapper you may have used is jQuery's `$(el).css()` method. Not only does it help normalize the interfaces to how styles can be applied between a number of browsers, there are plenty of good examples of this, including opacity.

```
/*
  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
*/

$('.container').css({ opacity: .5 });
```

The Facade Pattern

As we saw in earlier sections, the **Facade Pattern** is where an object provides a simpler interface to a larger (possibly more complex) body of code. Facades can be frequently found across the jQuery library and make methods both easier to use and understand, but also more readable. 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. You see, `$.ajax` offers a much simpler interface to a complex body of code that handles cross-browser XHR (XMLHttpRequest) as well as deferreds. While I could link you to the jQuery source, here's a cross-browser XHR implementation just so you can get an idea of how much easier this pattern makes our lives.

The Observer Pattern

Another pattern we've look at previously is the Observer (Publish/Subscribe) pattern, where a subject (the publisher), keeps a list of its dependants (subscribers), and notifies them automatically anytime something interesting happens.

jQuery actually comes with built-in support for a publish/subscribe-like system, which it calls custom events. In earlier versions of the library, access to these custom events was possible using `.bind()` (subscribe), `.trigger()` (publish) and `.unbind()` (unsubscribe), but in recent versions this can be done using `.on()`, `.trigger()` and `.off()`.

Below 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');
```

For those that prefer to use the conventional naming scheme for the Observer pattern, Ben Alman created a simple wrapper around the above methods which gives you access to `$.publish()`, `$.subscribe` and `$.unsubscribe` methods. I've previously linked to them earlier in the book, but you 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));
```

Finally, in recent versions of jQuery, a multi-purpose callbacks object (`$.Callbacks`) was made available to enable users to write new solutions based on callback lists. One such solution it's possible 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.
/*
output:
hello world!
fn2 says: hello world!
woo! mail!
*/
```

The Iterator Pattern

The Iterator Pattern is a design pattern where 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 `$(el).each()` iterator, you are actually able to use the underlying code behind `$.each()` to iterate through a collection, without needing to see or understand the code working behind the scenes that's providing this capability. This is a pattern similar to the facade, except it deals explicitly with iteration.

```
 $.each(['john','dave','rick','julian'], function(index, value) {
  console.log(index + ': ' + value);
});

$('li').each(function(index) {
  console.log(index + ': ' + $(this).text());
});
```

The Strategy Pattern

The Strategy Pattern is a pattern where a script may select a particular algorithm at runtime. The purpose of this pattern is that it's able to provide a way to clearly define families of algorithms, encapsulate each as an object and make them easily interchangeable. You could say that the biggest benefit this pattern offers is that it allows algorithms to vary independent of

the clients that utilize them.

An example of this is where jQuery's `toggle()` allows you to bind two or more handlers to the matched elements, to be executed on alternate clicks.The strategy pattern allows for alternative algorithms to be used independent of the client internal to the function.

```
$('button').toggle(function(){
    console.log('path 1');
},
function(){
    console.log('path 2');
});
```

The Proxy Pattern

The Proxy Pattern - a proxy is basically a class that functions as an interface to something else: a file, a resource, an object in memory, something else that is difficult to duplicate etc. jQuery's `.proxy()` method takes as input a function and returns a new one that will always have a particular context - it ensures that the value of `this` in a function is the value you desire. This is parallel to the idea of providing an interface as per the proxy pattern.

One example of where this is useful is when you're making use of a timer inside a `click` handler. Say we have the following handler:

```
$('button').on('click', function(){
 // Within this function, 'this' refers to the element that was clicked
  $(this).addClass('active');
});
```

However, say we wished to add in a delay before the `active` class was added. One thought that comes to mind is using `setTimeout` to achieve this, but there's a slight problem here: whatever function is passed to `setTimeout` will have a different value for `this` inside that function (it will refer to `window` instead).

```
$('button').on('click', function(){
  setTimeout(function(){
    // 'this' doesn't refer to our element!
    $(this).addClass('active');
  });
```

```
});
```

To solve this problem, we can use `$.proxy()`. By calling it with the function and value we would like assisnged to `this` it will actally return a function that retains the value we desire. 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.
});
```

The Builder Pattern

The Builder Pattern's general idea is that it abstracts the steps involved in creating objects so that different implementations of these steps have the ability to construct different representations of objects. Below are examples of how jQuery utilizes this pattern to allow you to dynamically create new elements.

```
$('<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');
```

The Prototype Pattern

As we've seen, the **Prototype Pattern** is used when objects are created based on a template of an existing object through cloning. Essentially this pattern is used to avoid creating a new object in a more conventional manner where this process may be expensive or overly complex.

In terms of the jQuery library, your first thought when cloning is mentioned might be the `.clone()` method. Unfortunately this only clones DOM elements but if we want to clone

JavaScript objects, this can be done using the `$.extend()` method as follows:

```
var myOldObject = {};

// Create a shallow copy
var myNewObject = jQuery.extend({}, myOldObject);

// Create a deep copy
var myOtherNewObject = jQuery.extend(true, {}, myOldObject);
```

This pattern has used many times in jQuery core (as well as in jQuery plugins) quite successfully. For those wondering what deep cloning might look like in JavaScript without the use of a library, Rick Waldron has an implementation you can use below (and tests available here).

```
function clone( obj ) {
  var val, length, i,
    temp = [];

  if ( Array.isArray(obj) ) {
    for ( i = 0, length = obj.length; i
```

Modern Modular JavaScript Design Patterns

The Importance Of Decoupling Your Application

In the world of modern 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. As you probably know, loose coupling facilitates easier maintainability of apps by removing *dependencies* where possible. When this is implemented efficiently, its quite easy to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages however, 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 haven't required great thought until more recent years where 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 your architecture. There's also no clean way to handle dependency management without some manual effort or third party tools.

Whilst native solutions to these problems will be arriving in ES Harmony (the next version of JavaScript), the good news is that writing modular JavaScript has never been easier and you 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, 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. In order 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 article, but I can recommend reading John Hann's article about curl.js and James Burke's RequireJS API documentation 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 you 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 A Format For Writing Modular JavaScript In The Browser

The overall goal for the AMD (Asynchronous Module Definition) 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 where 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 towards 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 concensus, further development of the format moved to the amdjs group.

Today it's embraced by projects including Dojo (1.7), MooTools (2.0), Firebug (1.8) and even jQuery (1.7). 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.

Note: There was a time when the proposal was referred to as Modules Transport/C, however as the spec wasn't geared for 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 two key concepts you need to be aware of here are the idea of a `define` met hod for facilitating module definition and a `require` method for handling depen dency loading. *define* is used to define named or unnamed modules based on the proposal using the following signature:

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

As you can tell by the inline comments, the `module_id` is an optional argument w hich is typically only required when non-AMD concatenation tools are being us ed (there may be some other edge cases where it's useful too). When this argu ment is left out, we call the module 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 cod e is more portable, it can be easily moved to other locations (or around the file-system) without needing to alter the code itself or change its ID. The `mo dule_id` is equivalent to folder paths in simple packages and when not used in packages. Developers can also run the same code on multiple environments just by using an AMD optimizer that works with a CommonJS environment such as r.j s.

Back to the define signature, the dependencies argument represents an array o f dependencies which are required by the module you are defining and the thir d argument ('definition function' or 'factory function') is a function that's executed to instantiate your module. A barebone module could be defined as f

ollows:

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 example 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 as it's relatively flexible 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 you wish to dynamically fetch dependencies. An example of its usage is:

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();
});
```

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
    };
});
```

Understanding AMD: plugins

The following is an example of defining an AMD-compatible plugin:

```
// 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 some fun template stuff here.
    }
});
```

Note: 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 you approach your build, 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.

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();
});
```

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).

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();
});
```

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();
});
```

Why Is AMD A Better Choice For Writing Modular JavaScript?

- 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 stand-alone modules and dependencies they may have.
- Module definitions are encapsulated, helping us to avoid pollution of the global namespace.
- Works better than some alternative solutions (eg. CommonJS, which we'll be looking at shortly). Doesn't have issues with cross-domain, local or debugging and doesn't have a reliance 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.

AMD Modules With Dojo

Defining AMD-compatible modules using Dojo is fairly straight-forward. As per above, define any module dependencies in an array as the first argument and provide a callback (factory) which will execute the module once the dependencies have been loaded. e.g:

```
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 that you may be used to using.

For those wondering about module referencing, there are some interesting gotchas 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 Dojo 1.6 build system - it really only works for AMD-compliant loaders. e.g:

```
define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){
    var cookieValue = cookie("cookieName");
    new Tree(...);
});
```

This has many advances 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 your applications.

Note: Although Dojo 1.6 doesn't officially support user-based AMD modules (nor asynchronous loading), it's possible to get this working with Dojo using a number of different script loaders. At present, all Dojo core and Dijit modules have been transformed to the AMD syntax and improved overall AMD support will likely land between 1.7 and 2.0.

The final gotcha to be aware of is that if you wish to continue using the 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, diji
t){
    var cookieValue = dojo.cookie("cookieName");
    new dijit.Tooltip(...);
});
```

AMD Module Design Patterns (Dojo)

If you've followed any of my previous posts on the benefits of design patterns, you'll know that they can be highly effective in improving how we approach structuring solutions to common development problems. John Hann recently gave an excellent presentation about AMD module design patterns covering the Singleton, Decorator, Mediator and others. I highly recommend checking out his slides if you get a chance.

Some samples of these patterns can be found below:

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; // makes `new` optional
    };
});


// decorator consumer
// a consumer for mylib/UpdatableObservable
```

```
define(['mylib/UpdatableObservable'], function ( makeUpdatable ) {
    var observable, updatable, someItem;
    // ... here be code to get or create `observable`

    // ... make the observable store updatable
    updatable = makeUpdatable(observable); // `new` is optional!

    // ... later, when a cometd message arrives with new data item
    updatable.updated(updatedItem);
});
```

Adapter pattern

```
// 'mylib/Array' adapts `each` function to mimic jQuery's:
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
The Basics

Unlike Dojo, jQuery really only comes with one file, however given the plugin-based nature of the library, we can demonstrate how straight-forward 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 });

        return {};
        // What we return can be used by other modules
    });
```

There is however something missing from this example and it's 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) which 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 their jQuery version leaking into the global space, they 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 () {};
});
```

Smarter jQuery Plugins

I've recently discussed some ideas and examples of how jQuery plugins could be written using Universal Module Definition (UMD) patterns here. UMDs define modules that can work on both the client and server, as well as with all popular script loaders available at the moment. Whilst this is still a new area with a lot of concepts still being finalized, feel free to look at the code samples in the section title *AMD && CommonJS* below and let me know if you feel there's anything we could do better.

What Script Loaders & Frameworks Support AMD?

In-browser:
Server-side:

- RequireJS http://requirejs.org
- PINF http://github.com/pinf/loader-js

AMD Conclusions

The above are very trivial examples of just how useful AMD modules can truly be, but they hopefully provide a foundation for understanding how they work.

You may be interested to know that many visible large applications and companies currently use AMD modules as a part of their architecture. These include IBM and the BBC iPlayer, which highlight just how seriously this format is being considered by developers at an enterprise-level.

For more reasons why many developers are opting to use AMD modules in their applications, you may be interested in this post by James Burke.

CommonJS A Module Format Optimized For The Server

CommonJS are a volunteer working group which aim to design, prototype and standardize JavaScript APIs. To date they've attempted to ratify standards for both modules and packages.

The CommonJS module proposal specifies a simple API for declaring modules server-side and unlike AMD attempts to cover a broader set of concerns such as io, filesystem, promises and more.

Getting Started

From a structure perspective, a CommonJS module is a reusable piece of JavaScript which exports specific objects made available to any dependent code - there are typically no function wrappers around such modules (so you won't see `define` used here for example).

At a high-level they 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.

Understanding CommonJS: require() and exports

```
// package/lib is a dependency we require
var lib = require('package/lib');

// some behaviour for our module
function foo(){
    lib.log('hello world!');
}

// export (expose) foo to other modules
exports.foo = foo;
```

Basic consumption of exports

```
// define more behaviour we would like to expose
function foobar(){
        this.foo = function(){
                console.log('Hello foo');
        }

        this.bar = function(){
                console.log('Hello bar');
        }
}

// expose foobar to other modules
```

```
exports.foobar = foobar;


// an application consuming 'foobar'

// access the module relative to the path
// where both usage and module files exist
// in the same directory

var foobar = require('./foobar').foobar,
    test   = new foobar();

test.bar(); // 'Hello bar'
```

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!!''
}
```

What Loaders & Frameworks Support CommonJS?

In-browser:
Server-side:

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 which 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 which may be used everywhere. Modules which 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 plugins and can define more granular things like constructors and functions this makes sense. CommonJS modules are only able to define objects which can be tedious to work with if you'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 are

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`.

AMD && CommonJS Competing, But Equally Valid Standards

Whilst this section has placed more emphasis on using AMD over CommonJS, the reality is that both formats are valid and have a use.

AMD adopts a browser-first approach to development, opting for asynchronous behaviour and simplified backwards compatability 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 behaviour, no global *baggage* as John Hann would refer to it as and it 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 you of the `define()` wrapper that AMD enforces. CommonJS modules however only support objects as modules.

Although the idea of yet another module format may be daunting, you may be interested in some samples of work on hybrid AMD/CommonJS and Univeral AMD/CommonJS modules.

Basic AMD Hybrid Format (John Hann)

```
define( function (require, exports, module){

    var shuffler = require('lib/shuffle');

    exports.randomize = function( input ){
        return shuffler.shuffle(input);
    }
});
```

Note: this is basically the 'simplified CommonJS wrapper' that is supported in the AMD spec.

AMD/CommonJS Universal Module Definition (Variation 2, UMDjs)

```
/**
```

```
 * exports object based version, if you need to make a
 * circular dependency or need compatibility with
 * commonjs-like environments that are not Node.
 */
(function (define) {
    //The 'id' is optional, but recommended if this is
    //a popular web library that is used mostly in
    //non-AMD/Node environments. However, if want
    //to make an anonymous module, remove the 'id'
    //below, and remove the id use in the define shim.
    define('id', function (require, exports) {
        //If have dependencies, get them here
        var a = require('a');

        //Attach properties to exports.
        exports.name = value;
    });
}(typeof define === 'function' && define.amd ? define : function (id, factory
) {
    if (typeof exports !== 'undefined') {
        //commonjs
        factory(require, exports);
    } else {
        //Create a global function. Only works if
        //the code does not have dependencies, or
        //dependencies fit the call pattern below.
        factory(function(value) {
            return window[value];
        }, (window[id] = {}));
    }
}));
```

Extensible UMD Plugins With (Variation by myself and Thomas Davis).

core.js

```
// Module/Plugin core
// Note: the wrapper code you 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.

;(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 plugins highlighting elements different
  // colors
  module.highlight = function(el,strColor){
    // this module uses jQuery, however plain old JavaScript
    // or say, Dojo could be just as easily used.
    if(this.jQuery){
      jQuery(el).css('background', strColor);
    }
  }
  return {
      highlightAll:function(){
        module.highlight('div', module.highlightColor);
      }
  };

});
```

myExtension.js

```
;(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 (wind
ow)



        // account for for flat-file/global module extensions
        var obj = null;
        var namespaces = name.split(".");
        var scope = (this.jQuery || this.ender || this.$ || this);
        for (var i = 0; i
```

app.js

```
$(function(){

    // the plugin 'core' is exposed under a core namespace in
    // this example which we first cache
    var core = $.core;

    // use then use some of the built-in core functionality to
    // highlight all divs in the page yellow
    core.highlightAll();

    // access the plugins (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 plugin loaded in after it

    // Set the last div to the 'errorColor' property defined in
    // our core module/plugin. If you review the code further down
    // you'll see how easy it is to consume properties and methods
    // between the core and other plugins
    core.plugin.setRed('div:last');
});
```

ES Harmony Modules Of The Future

TC39, the standards body charged with defining the syntax and semantics of EC MAScript and its future iterations is composed of a number of very intelligen t developers. Some of these developers (such as Alex Russell) have been keepi ng a close eye on the evolution of JavaScript usage for large-scale developme nt over the past few years and are acutely aware of the need for better langu age features for writing more modular JS.

For this reason, there are currently proposals for a number of exciting addit ions to the language including flexible modules that can work on both the cli ent and server, a module loader and more. In this section, I'll be showing yo u some code samples of the syntax for modules in ES.next so you can get a tas te of what's to come.

Note: Although Harmony is still in the proposal phases, you can already try out (partial) features of ES.next that address native support for writing mod ular JavaScript thanks to Google's Traceur compiler. To get up and running wi th Traceur in under a minute, read this getting started guide. There's also a JSConf presentation about it that's worth looking at if you're interested in learning more about the project.

Modules With Imports And Exports

If you've read through the sections on AMD and CommonJS modules you may be fa miliar 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 dep endencies being specified using an `import` keyword. `export` isn't greatly differe nt to what we might expect and I think many developers will look at the code

```
below and instantly 'get' it.
```

- **import** declarations bind a module's 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 however can't export modules that have been defined elsewhere. You may also rename exports so their external name differs 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 API wrapper) making it simplistic to load modules in from external locations. Here's an example of us 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 of us 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 server-oriented, the module system proposed for ES.next isn't just constrained to looking at modules in the browser. Below for examples, you 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 & 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 desugars to the same prototypal behavior.

In Harmony, classes come as part of the language along with constructors and (finally) some sense of true privacy. In the following examples, I've included some inline comments to help you understand how classes are structured, but you 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
```

ES Harmony Conclusions

As you can see, ES.next is coming with some exciting new additions. Although Traceur can be used to an extent to try our such features in the present, remember that it may not be the best idea to plan out your system to use Harmony (just yet). There are risks here such as specifications changing and a potential failure at the cross-browser level (IE9 for example will take a while to die) so your best bets until we have both spec finalization and coverage are AMD (for in-browser modules) and CommonJS (for those on the server).

Conclusions And Further Reading A Review

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 (classical) module pattern alone including: avoiding a need for developers to create global variables for each module they create, better support for static and dynamic dependency management, improved compatibility with script loaders, better (optional) compatibility for modules on the server and more.

In short, I recommend trying out what's been suggested today as these formats offer a lot of power and flexibility that can help when building applications based on many reusable blocks of functionality.

Bonus: jQuery Plugin Design Patterns

While well-known JavaScript design patterns can be extremely useful, another side of development could benefit from its own set of design patterns are jQuery plugins. The official jQuery plugin authoring guide offers a great starting point for getting into writing plugins and widgets, but let’s take it further.

Plugin development has evolved over the past few years. We no longer have just one way to write plugins, but many. In reality, certain 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 plugins more like modules (similar to the module pattern) or use a more

formal module format such as AMD (asynchronous module definition). Some might want their plugins to harness the power of prototypal inheritance. Some might want to use custom events or pub/sub to communicate from plugins to the rest of their app. And so on.

I began to think about plugin patterns after noticing a number of efforts to create a one-size-fits-all jQuery plugin boilerplate. While such a boilerplate is a great idea in theory, the reality is that we rarely write plugins in one fixed way, using a single pattern all the time.

Let's assume that you've tried your hand at writing your own jQuery plugins at some point and you're comfortable putting together something that works. It's functional. It does what it needs to do, but perhaps you feel it could be structured better. Maybe it could be more flexible or could solve more issues. If this sounds familiar and you aren't sure of the differences between many of the different jQuery plugin patterns, then you might find what I have to say helpful.

My advice won't provide solutions to every possible pattern, but it will cover popular patterns that developers use in the wild.

Note: This section is targeted at intermediate to advanced developers. If you don't feel you're ready for this just yet, I'm happy to recommend the official jQuery Plugins/Authoring guide, Ben Alman's plugin style guide and Remy Sharp's "Signs of a Poorly Written jQuery Plugin."

Patterns

jQuery plugins have very few defined rules, which one of the reasons for the incredible diversity in how they're implemented. At the most basic level, you

can write a plugin simply by adding a new function property to jQuery's `$.fn` object, as follows:

```
$.fn.myPluginName = function() {
    // your plugin logic
};
```

This is great for compactness, but the following would be a better foundation to build on:

```
(function( $ ){
  $.fn.myPluginName = function() {
    // your plugin logic
  };
})( jQuery );
```

Here, we've wrapped our plugin 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, thus ensuring 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 `$.extend`, which enables you to define multiple functions at once and which sometimes make more sense semantically:

```
(function( $ ){
    $.extend($.fn, {
        myplugin: function(){
            // your plugin logic
        }
    });
```

```
})( jQuery );
```

We could do a lot more to improve on all of this; and the first complete pattern we'll be looking at today, the lightweight pattern, covers some best practices that we can use for basic everyday plugin development and that takes into account common gotchas to look out for.

Note

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 practices are best.

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 that you can read up on their individual work if interested.

A Lightweight Start

Let's begin our look at patterns with something basic that follows best practices (including those in the jQuery plugin-authoring guide). This pattern is ideal for developers who are either new to plugin development or who just want to achieve something simple (such as a utility plugin). This lightweight start uses the following:

- Common best practices, such as a semi-colon before the function's invocation; `window, document, undefined` passed in as arguments; and adherence to the jQuery core style guidelines.
- A basic defaults object.
- A simple plugin 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.

```
/*!
 * 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 plugins
// 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 your plugin).

    // Create the defaults once
    var pluginName = 'defaultPluginName',
        defaults = {
            propertyName: "value"
        };

    // The actual plugin 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 plugin
        this.options = $.extend( {}, defaults, options) ;

        this._defaults = defaults;
        this._name = pluginName;

        this.init();
    }

    Plugin.prototype.init = function () {
        // Place initialization logic here
        // You already have access to the DOM element and
        // the options via the instance, e.g. this.element
        // and this.options
    };

    // A really lightweight plugin 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

“Complete” Widget Factory

While the authoring guide is a great introduction to plugin development, it doesn’t offer a great number of conveniences for obscuring away from common 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 you build complex, stateful plugins based on object-oriented principles. It also eases communication with your plugin’s instance, obfuscating a number of the repetitive tasks that you would have to code when working with basic plugins.

In case you haven’t come across these before, stateful plugins keep track of their current state, also allowing you to change properties of the plugin 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 you’re looking for further guidance on structure beyond this template, you won’t have to look beyond the jQuery UI repository.

Back to patterns. This jQuery UI boilerplate does the following:

- Covers almost all supported default methods, including triggering events.
- Includes comments for all of the methods used, so that you’re never unsure of where logic should fit in your plugin.

```
/*!
 * 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 your 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 (eg. 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 you 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 plugin 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 plugin user
    // can subscribe to
    // signature: _trigger( "callbackName" , [eventObject],
    // [uiObject] )
    // eg. 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 = $('#elem').widgetName({
  foo: false
});

instance.widgetName('methodB');
```

Further Reading

Namespacing And Nested Namespacing

Namespacing your code is a way to avoid collisions with other objects and variables in the global namespace. They're important because you want to safeguard your plugin from breaking in the event that another script on the page uses the same variable or plugin names as yours. As

a good citizen of the global namespace, you must also do your 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 your namespace, you 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 plugin.

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 give you everything you 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 your initialization code here
        };

        // Sample Function, Uncomment to use
        // base.functionName = function( paramaters ){
        //
        // };
        // 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

Custom Events For Pub/Sub (With The Widget factory)

You may have used the Observer (Pub/Sub) pattern in the past to develop asynchronous JavaScript web applications. The basic idea here is that elements will publish event notifications when something interesting occurs in your application. Other elements then subscribe to or listen for these events and respond accordingly. This results in the logic for your application being significantly more decoupled (which is always good).

In jQuery, we have this idea that custom events provide a built-in means to implement a publish and subscribe system that's quite similar to the Observer pattern. So, `bind('eventType')` is functionally equivalent to performing `subscribe('eventType')`, and `trigger('eventType')` is roughly equivalent to `publish('eventType')`.

Some developers might consider the jQuery event system as having too much overhead to be used as a publish and subscribe system, but it's been architected to be both reliable and robust for most use cases. In the following jQuery UI widget factory template, we'll implement a basic custom event-based pub/sub pattern that allows our plugin to subscribe to event notifications from the rest of our application, which publishes 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.bind( "myEventStart", function( e ) {
                console.log("event start");
            });

            //subscribe to 'myEventEnd'
            self.element.bind( "myEventEnd", function( e ) {
```

```
                console.log("event end");
            });

            //unsubscribe to 'myEventStart'
            //self.element.unbind( "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

In JavaScript, we don't have the traditional notion of classes that you would find in other classical programming languages, but we do have prototypal inheritance. With prototypal inheritance, an object inherits from another object. And we can apply this concept to jQuery plugin development.

Alex Sexton and Scott Gonzalez have looked at this topic in detail. In sum, they found that for organized modular development, clearly separating the object that defines the logic for a plugin from the plugin-generation process itself can be beneficial. The benefit is that testing your plugin's code becomes easier, and you can also adjust the way things work behind the scenes without altering the way that any object APIs you've implemented are used.

In Sexton's previous post on this topic, he implements a bridge that enables you to attach your general logic to a particular plugin, which we've implemented in the template below. Another advantage of this pattern is that you don't have to constantly repeat the same plugin 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 that you want
// 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.$elem.html('<h1>'+this.options.name+'</h1>');
  },
  myMethod: function( msg ){
    // You have direct access to the associated and cached
    // jQuery element
    // this.$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 plugin 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 instance = $('#elem').data('myobj');
instance.myMethod('I am a method');
```

Further Reading

jQuery UI Widget Factory Bridge

If you liked the idea of generating plugins 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 jQuery's API, providing a more built-in solution to

achieving object-based plugin definition. Effectively, we're able to create stateful plugins 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, look at the comments in the boilerplate 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 plugin goes into _init
// This fires when your 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
});

// your widget instance exists in the elem's data
console.log(instance.data("foo").element); // => #elem element

// bridge allows you to call public methods...
instance.foo("publicFunction"); // => "public method"

// bridge prevents calls to internal methods
instance.foo("_privateFunction"); // => #elem element
```

Further Reading

- "Using $.widget.bridge Outside of the Widget Factory," Eric Hynds

jQuery Mobile Widgets With The Widget factory

jQuery mobile is a 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, you 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 plugin and widget development, as seen in some of the core jQuery mobile widgets used in the official library suite. What's interesting here is that even though there are very small, subtle differences in writing a "mobile"-optimized widget, if you're familiar with using the jQuery UI Widget Factory, you should be able to start writing these right away.

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.
- You'll notice in `_create()` a guide on how the official jQuery mobile widgets handle element selection, opting for a role-based approach that better fits the jQM mark-up. 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 jQM pages.
- Guidelines are also provided in comment form for applying your plugin methods on `pagecreate` as well as for selecting the plugin 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 your choice
    //here 'mobile' has been used in the first parameter
    $.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 you 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 plugin 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 plugin user can
    // subscribe to
    //signature: _trigger( "callbackName" , [eventObject],
    //  [uiObject] )
    // eg. 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" plugin dispatches a "create" event when a jQuery Mobile page (found via data-role=page attr) is first initialized.We can listen for that event (called "pagecreate") and run our plugin automatically whenever a new page is created.

```
$(document).bind("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 plugin on them.
    // Here's how we'd call our "foo" plugin 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 configurab
le
    // data-attribute namespace
    $(e.target).find(":jqmData(role='foo')").foo(options);
});
```

That's it. Now you can simply reference the script containing your widget and pagecreate binding in a page running jQuery Mobile site, and it will automatically run like any other jQuery Mobile plugin.

RequireJS And The jQuery UI Widget Factory

RequireJS is a 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 plugin); it simplifies the process of combining scripts via its excellent optimizer; and it provides the means for defining module dependencies on a per-module basis.

James Burke has written a comprehensive set of tutorials on getting started with RequireJS. But what if you're already familiar with it and would like to wrap your jQuery UI widgets or plugins in a RequireJS-compatible module wrapper?.

In the boilerplate pattern below, we demonstrate how a compatible widget can be defined that does the following:

- Allows the definition of widget module dependencies, building on top of the previous jQuery UI boilerplate presented earlier;
- Demonstrates one approach to passing in HTML template assets for creating templated widgets with jQuery (in conjunction with the jQuery tmpl plugin) (View the comments in `_create()`.)
- Includes a quick tip on adjustments that you can make to your widget module if you wish to later pass it through 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 you 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 you can construct the widget like so:



//ao.myWidget.js file:
define("ao.myWidget", ["jquery", "text!templates/asset.html", "jquery-ui.cust
om.min","jquery.tmpl"], function ($, assetHtml) {

    // define your widget under a namespace of your 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 you 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();
            //this.element.tmpl(assetHtml).appendTo(this.content);
        },

        // Destroy an instantiated plugin and clean up modifications
        // that the widget has made to the DOM
        destroy: function () {
            //t his.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 plugin 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 );
}

//somewhere assetHtml would be used for templating, depending
// on your choice.
```

```
    });
});
```

Usage:

index.html:

```
<script data-main="scripts/main" src="http://requirejs.org/docs/release/1.0.1
/minified/require.js"></script>
```

main.js

```
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/jq
uery-ui.min',
        'boilerplate': '../patterns/jquery.widget-factory.requirejs.boilerpla
te'
    }
}, ['require', 'jquery', 'jqueryui', 'boilerplate'],
function (req, $) {

    $(function () {

        var instance = $('#elem').myWidget();
        instance.myWidget('methodB');

    });
});
```

Further Reading

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 your plugin. The way you're probably familiar with defining plugin options is to pass through an object literal of defaults to `$.extend`, as demonstrated in our basic plugin boilerplate.

If, however, you're working with a plugin with many customizable options that you would like

users to be able to override either globally or on a per-call level, then you can structure things a little differently.

Instead, by referring to an options object defined within the plugin namespace explicitly (for example, `$fn.pluginName.options`) and merging this with any options passed through to the plugin when it is initially invoked, users have the option of either passing options through during plugin initialization or overriding options outside of the plugin (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 plugin. 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 plugin 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.
    // eg. $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, Ben Alman

A Highly Configurable And Mutable Plugin

Like Alex Sexton's pattern, the following logic for our plugin isn't nested in a jQuery plugin itself. We instead define our plugin's logic using a constructor and an object literal defined on its prototype, using jQuery for the actual instantiation of the plugin object.

Customization is taken to the next level by employing two little tricks, one of which you'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 plugin 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.

You don't see the latter option in the wild too often, but it can be a significantly cleaner solution (as long as you don't mind the inline approach). If you're wondering where this could be useful, imagine writing a draggable plugin for a large set of elements. You could go about customizing their options like this:

```
javascript
$('.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:

```
javascript
$('.items').draggable();

html
<li class="item" data-plugin-options='{"defaultPosition":"top-left"}'></div>
<li class="item" data-plugin-options='{"defaultPosition":"bottom-left"}'></di
v>
```

And so on. You may well have a preference for one of these approaches, but it is another potentially useful pattern to be 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 plugin logic
// hasn't been nested in a jQuery plugin. Instead, we just use
// jQuery for its instantiation.

;(function( $, window, document, undefined ){

  // our plugin 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 plugin 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 plugin 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() {
    // eg. 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

UMD: AMD And CommonJS-Compatible Modules For Plugins

Whilst many of the plugin and widget patterns presented above are acceptable for general use, they aren't without their caveats. Some require jQuery or the jQuery UI Widget Factory to be present in order to function, while only a few could be easily adapted to work well as globally compatible modules in both the browser and other environments.

We've alredy explored both AMD and CommonJS in the last chapter, but imagine how useful it would be if we could define and load plugin modules compatible with AMD, CommonJS and other standards that are also compatible with different environments (client-side, server-side and beyond).

To provide a solution for this problem, a number of developers including James Burke, myself, Thomas Davis and Ryan Florence have been working on an effort known as UMD (or Universal Module Definition). The goal of our efforts has been to provide a set of agreed upon patterns for plugins that can work in all environments. At present, a number of such boilerplates have been completed and are available on the UMD group repo https://github.com/umdjs/umd.

One such pattern we've worked on for jQuery plugins appears below and has the following features:

- A core/base plugin is loaded into a `$.core` namespace, which can then be easily extended using plugin extensions via the namespacing pattern. Plugins loaded via script tags automatically populate a `plugin` namespace under `core` (i.e. `$.core.plugin.methodName()`).
- The pattern can be quite nice to work with because plugin 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 isn't required at all to make this pattern fully function.

usage.html

```
<script type="text/javascript" src="http://code.jquery.com/jquery-1.7.1.min.j
s"></script>
<script type="text/javascript" src="pluginCore.js"></script>
<script type="text/javascript" src="pluginExtension.js"></script>

<script type="text/javascript">

$(function(){

    // Our plugin '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 plugins (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 plugin loaded in after it

    // Set the last div to the 'errorColor' property defined in
    // our core module/plugin. If you review the code further down,
    // you'll see how easy it is to consume properties and methods
    // between the core and other plugins
    core.plugin.setRed('div:last');
});

</script>
```

pluginCore.js

```
// Module/Plugin core
// Note: the wrapper code you 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 plugins 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 (wind
ow)



        // account for for flat-file/global module extensions
        var obj = null;
        var namespaces = name.split(".");
        var scope = (this.jQuery || this.ender || this.$ || this);
        for (var i = 0; i
```

Whilst work on improving these patterns is ongoing, please do feel free to check out the patterns suggested to date as you may find them helpful.

Further Reading

What Makes A Good Plugin Beyond Patterns?

At the end of the day, patterns are just one aspect of plugin development. And before we wrap up, here are my criteria for selecting third-party plugins, which will hopefully help developers write them.

Quality

Do your best to adhere to best practices with both the JavaScript and jQuery that you write. Are your solutions optimal? Do they follow the jQuery Core Style Guidelines? If not, is your code at least relatively clean and readable?

Compatibility

Which versions of jQuery is your plugin compatible with? Have you tested it with the latest builds? If the plugin was written before jQuery 1.6, then it might have issues with attributes, because the way we approach them changed with 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 towards a better way of doing things. I'd like to see plugin authors update their code when necessary or, at a minimum, test their plugins with new versions to make sure everything works as expected.

Reliability

Your plugin should come with its own set of unit tests. Not only do these prove your plugin actually works, but they can also improve the design without breaking it for end users. I consider unit tests essential for any serious jQuery plugin 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 Jorn Zaefferer.

Performance

If the plugin needs to perform tasks that require a lot of computing power or that heavily manipulates the DOM, then you should follow best practices that minimize this. Use jsPerf.com to test segments of your code so that you're aware of how well it performs in different browsers before releasing the plugin.

Documentation

If you intend for other developers to use your plugin, ensure that it's well documented. Document your API. What methods and options does the plugin support? Does it have any gotchas that users need to be aware of? If users cannot figure out how to use your plugin, they'll likely look for an alternative. Also, do your best to comment the code. This is by far the best gift you could give to other developers. If someone feels they can navigate your code base well enough to fork it or improve it, then you've done a good job.

Likelihood of maintenance

When releasing a plugin, estimate how much time you'll have to devote to maintenance and support. We all love to share our plugins with the community, but you need to set expectations for your ability to answer questions, address issues and make improvements. This can be done simply by stating your intentions for maintenance in the *README* file, and let users decide whether to make fixes themselves.

In this section, we've explored several time-saving design patterns and best practices that can be employed to improve your plugin development process. Some are better suited to certain use cases than others, but I hope that the code comments that discuss the ins and outs of these variations on popular plugins and widgets were useful.

Remember, when selecting a pattern, be practical. Don't use a plugin pattern just for the sake of it; rather, spend some time understanding the underlying structure, and establish how well it solves your problem or fits the component you're trying to build. Choose the pattern that best suits your needs.

Conclusions

That's it for this introduction to the world of design patterns in JavaScript - I hope you've found it useful. The contents of this book should hopefully have given you sufficient information to get started using the patterns covered in your day-to-day projects.

Design patterns make it easier to reuse successful designs and architectures. It's important for every developer to be aware of design patterns but it's also essential to know how and when to use them. Implementing the right patterns intelligently can be worth the effort but the opposite is also true. A badly implemented pattern can yield little benefit to a project.

Also keep in mind that it is not the number of patterns you implement that's important but how you choose to implement them. For example, don't choose a pattern just for the sake of using 'one' but rather try understanding the pros and cons of what particular patterns have to offer and make a judgement based on it's fitness for 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 but also those that cover specific languages.

I'm happy to recommend:

If you've managed to absorb most of the information in my book, I think you'll find reading these the next logical step in your learning process (beyond trying out some pattern examples for yourself of course).

Thanks for reading *Essential JavaScript Design Patterns*. For more educational material on learning JavaScript, please feel free to read more from me on my blog http://addyosmani.com or on Twitter @addyosmani.

References

Original URL:

http://addyosmani.com/resources/essentialjsdesignpatterns/book/?43243242424