

THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Pro JavaScript™ Design Patterns

*The essentials of object-oriented
JavaScript™ programming*

Ross Harmes and Dustin Diaz

apress®

Pro JavaScript™ Design Patterns

Copyright © 2008 by Ross Harmes and Dustin Diaz

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-908-2

ISBN-10 (pbk): 1-59059-908-X

ISBN-13 (electronic): 978-1-4302-0495-4

ISBN-10 (electronic): 1-4302-0495-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. Apress Inc. is not affiliated with Sun Microsystems Inc., and this book was written without endorsement from Sun Microsystems Inc.

Lead Editors: Chris Mills, Tom Welsh

Technical Reviewer: Simon Willison

Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jason Gilmore, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Richard Dal Porto

Copy Editor: Jennifer Whipple

Associate Production Director: Kari Brooks-Copony

Production Editor: Kelly Winkvist

Compositor and Artist: Kinetic Publishing Services, LLC

Proofreader: Dan Shaw

Indexer: Julie Grady

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.

Contents

About the Authors	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi

PART 1 ■ ■ ■ Object-Oriented JavaScript

■ CHAPTER 1	Expressive JavaScript	3
	The Flexibility of JavaScript	3
	A Loosely Typed Language	6
	Functions As First-Class Objects	6
	The Mutability of Objects	8
	Inheritance	9
	Design Patterns in JavaScript	9
	Summary	10
■ CHAPTER 2	Interfaces	11
	What Is an Interface?	11
	Benefits of Using Interfaces	11
	Drawbacks of Using Interfaces	12
	How Other Object-Oriented Languages Handle Interfaces	12
	Emulating an Interface in JavaScript	14
	Describing Interfaces with Comments	14
	Emulating Interfaces with Attribute Checking	16
	Emulating Interfaces with Duck Typing	17
	The Interface Implementation for This Book	18
	The Interface Class	19
	When to Use the Interface Class	20
	How to Use the Interface Class	20
	Example: Using the Interface Class	21
	Patterns That Rely on the Interface	23
	Summary	23

CHAPTER 3	Encapsulation and Information Hiding	25
	The Information Hiding Principle	25
	Encapsulation vs. Information Hiding	26
	The Role of the Interface	26
	Basic Patterns	26
	Fully Exposed Object	27
	Private Methods Using a Naming Convention	30
	Scope, Nested Functions, and Closures	32
	Private Members Through Closures	33
	More Advanced Patterns	35
	Static Methods and Attributes	35
	Constants	37
	Singletons and Object Factories	38
	Benefits of Using Encapsulation	39
	Drawbacks to Using Encapsulation	39
	Summary	40
CHAPTER 4	Inheritance	41
	Why Do You Need Inheritance?	41
	Classical Inheritance	42
	The Prototype Chain	42
	The extend Function	43
	Prototypal Inheritance	45
	Asymmetrical Reading and Writing of Inherited Members	46
	The clone Function	48
	Comparing Classical and Prototypal Inheritance	49
	Inheritance and Encapsulation	49
	Mixin Classes	50
	Example: Edit-in-Place	52
	Using Classical Inheritance	52
	Using Prototypal Inheritance	55
	Using Mixin Classes	59
	When Should Inheritance Be Used?	62
	Summary	63
CHAPTER 5	The Singleton Pattern	65
	The Basic Structure of the Singleton	65
	Namespacing	66

A Singleton As a Wrapper for Page-Specific Code	68
A Singleton with Private Members	70
Using the Underscore Notation	70
Using Closures	71
Comparing the Two Techniques	74
Lazy Instantiation	75
Branching	78
Example: Creating XHR Objects with Branching	79
When Should the Singleton Pattern Be Used?	81
Benefits of the Singleton Pattern	81
Drawbacks of the Singleton Pattern	82
Summary	82
CHAPTER 6 Chaining	83
The Structure of a Chain	84
Building a Chainable JavaScript Library	86
Using Callbacks to Retrieve Data from Chained Methods	89
Summary	90

PART 2 ■ ■ ■ Design Patterns

CHAPTER 7 The Factory Pattern	93
The Simple Factory	93
The Factory Pattern	96
When Should the Factory Pattern Be Used?	99
Dynamic Implementations	99
Combining Setup Costs	99
Abstracting Many Small Objects into One Large Object	99
Example: XHR Factory	99
Specialized Connection Objects	101
Choosing Connection Objects at Run-Time	103
Example: RSS Reader	104
Benefits of the Factory Pattern	107
Drawbacks of the Factory Pattern	108
Summary	108

CHAPTER 8	The Bridge Pattern	109
	Example: Event Listeners	109
	Other Examples of Bridges	110
	Bridging Multiple Classes Together	111
	Example: Building an XHR Connection Queue	111
	Including the Core Utilities	112
	Including an Observer System	114
	Developing the Queue Skeleton	114
	Implementing the Queue	116
	Where Have Bridges Been Used?	122
	When Should the Bridge Pattern Be Used?	122
	Benefits of the Bridge Pattern	123
	Drawbacks of the Bridge Pattern	123
	Summary	123
CHAPTER 9	The Composite Pattern	125
	The Structure of the Composite	126
	Using the Composite Pattern	126
	Example: Form Validation	127
	Putting It All Together	133
	Adding Operations to FormItem	133
	Adding Classes to the Hierarchy	133
	Adding More Operations	136
	Example: Image Gallery	136
	Benefits of the Composite Pattern	139
	Drawbacks of the Composite Pattern	139
	Summary	140
CHAPTER 10	The Facade Pattern	141
	Some Facade Functions You Probably Already Know About	141
	JavaScript Libraries As Facades	142
	Facades As Convenient Methods	143
	Example: Setting Styles on HTML Elements	144
	Example: Creating an Event Utility	146
	General Steps for Implementing the Facade Pattern	147
	When Should the Facade Pattern Be Used?	148
	Benefits of the Facade Pattern	148
	Drawbacks of the Facade Pattern	148
	Summary	148

CHAPTER 11	The Adapter Pattern	149
	Characteristics of an Adapter	149
	Adapting Existing Implementations	150
	Example: Adapting One Library to Another	150
	Example: Adapting an Email API	152
	Wrapping the Webmail API in an Adapter	157
	Migrating from fooMail to dedMail	157
	When Should the Adapter Pattern Be Used?	158
	Benefits of the Adapter Pattern	158
	Drawbacks of the Adapter Pattern	158
	Summary	158
CHAPTER 12	The Decorator Pattern	159
	The Structure of the Decorator	159
	The Role of the Interface in the Decorator Pattern	163
	The Decorator Pattern vs. the Composite Pattern	163
	In What Ways Can a Decorator Modify Its Component?	164
	Adding Behavior After a Method	164
	Adding Behavior Before a Method	165
	Replacing a Method	166
	Adding New Methods	167
	The Role of the Factory	169
	Function Decorators	172
	When Should the Decorator Pattern Be Used?	173
	Example: Method Profiler	173
	Benefits of the Decorator Pattern	176
	Drawbacks of the Decorator Pattern	176
	Summary	177
CHAPTER 13	The Flyweight Pattern	179
	The Structure of the Flyweight	179
	Example: Car Registrations	179
	Intrinsic and Extrinsic State	180
	Instantiation Using a Factory	181
	Extrinsic State Encapsulated in a Manager	182
	Managing Extrinsic State	183
	Example: Web Calendar	183
	Converting the Day Objects to Flyweights	185
	Where Do You Store the Extrinsic Data?	186

Example: Tooltip Objects	186
The Unoptimized Tooltip Class	187
Tooltip As a Flyweight	188
Storing Instances for Later Reuse	190
When Should the Flyweight Pattern Be Used?	192
General Steps for Implementing the Flyweight Pattern	193
Benefits of the Flyweight Pattern	193
Drawbacks of the Flyweight Pattern	194
Summary	194
CHAPTER 14 The Proxy Pattern	197
The Structure of the Proxy	197
How Does the Proxy Control Access to Its Real Subject?	197
Virtual Proxy, Remote Proxy, and Protection Proxy	200
The Proxy Pattern vs. the Decorator Pattern	201
When Should the Proxy Be Used?	201
Example: Page Statistics	201
General Pattern for Wrapping a Web Service	205
Example: Directory Lookup	206
General Pattern for Creating a Virtual Proxy	210
Benefits of the Proxy Pattern	213
Drawbacks of the Proxy Pattern	213
Summary	214
CHAPTER 15 The Observer Pattern	215
Example: Newspaper Delivery	215
Push vs. Pull	216
Pattern in Practice	216
Building an Observer API	218
Delivery Method	219
Subscribe	219
Unsubscribe	220
Observers in Real Life	220
Example: Animation	221
Event Listeners Are Also Observers	222
When Should the Observer Pattern Be Used?	223
Benefits of the Observer Pattern	223
Drawbacks of the Observer Pattern	223
Summary	223

CHAPTER 16	The Command Pattern	225
	The Structure of the Command	225
	Creating Commands with Closures	227
	The Client, the Invoker, and the Receiver	227
	Using Interfaces with the Command Pattern	228
	Types of Command Objects	228
	Example: Menu Items	230
	The Menu Composites	231
	The Command Class	233
	Putting It All Together	234
	Adding More Menu Items Later On	235
	Example: Undo and Logging	235
	Implementing Undo with Nonreversible Actions By Logging Commands	239
	Logging Commands for Crash Recovery	242
	When to Use the Command Pattern	242
	Benefits of the Command Pattern	243
	Drawbacks of the Command Pattern	243
	Summary	244
CHAPTER 17	The Chain of Responsibility Pattern	245
	The Structure of the Chain of Responsibility	245
	Passing on Requests	251
	Implementing a Chain of Responsibility in an Existing Hierarchy	254
	Event Delegation	255
	When Should the Chain of Responsibility Pattern Be Used?	255
	Example: Image Gallery Revisited	256
	Using the Chain of Responsibility to Make Composites More Efficient	257
	Adding Tags to Photos	258
	Benefits of the Chain of Responsibility Pattern	261
	Drawbacks of the Chain of Responsibility Pattern	262
	Summary	262
INDEX		263

About the Authors



■ **ROSS HARMES** is a front-end engineer for Yahoo! in Sunnyvale, California. Educated as an electrical and computer engineer, Ross quickly put down the soldering iron and oscilloscope and focused on the software aspect of his degree. After discovering that debugging memory leaks is not much fun, he dove into the muddy and turbulent waters of web programming. He has been happily swimming there ever since.

This is Ross's first book, but he has been publishing his stray thoughts online for years. These days his technical ramblings can be found at <http://techfoolery.com>.



■ **DUSTIN DIAZ** is a user interface engineer for Google in Mountain View, California. He enjoys writing JavaScript, CSS, and HTML, as well as making interactive and usable interfaces to inspire passionate users. Dustin has written articles for Vitamin and Digital Web Magazine, and posts regularly about web development at his site, <http://dustindiaz.com>.

About the Technical Reviewer

■ **SIMON WILLISON** is a consultant on client- and server-side web development and a cocreator of the Django web framework. Simon's interests include OpenID, unobtrusive JavaScript, and rapid application development. Before going freelance, Simon worked on Yahoo!'s Technology Development team, and prior to that at the *Lawrence Journal-World*, an award-winning local newspaper in Kansas. Simon maintains a popular web development weblog at <http://simonwillison.net/>.



Expressive JavaScript

JavaScript is one of the most popular and widely used languages in the world today. Because it is embedded in all modern browsers, it has an extraordinarily wide distribution. As a language, it is incredibly important in our daily lives, powering the websites that we go to and helping the Web to present a rich interface.

Why then do some still consider it to be a toy language, not worthy of the professional programmer? We think it is because people do not realize the full power of the language and how unique it is in the programming world today. JavaScript is a very expressive language, with some features that are uncommon to the C family of languages.

In this chapter we explore some of the features that make JavaScript so expressive. We look at how the language allows you to accomplish the same task in a number of different ways and how you can take alternative approaches to object-oriented programming by using concepts from functional programming. We discuss why you should use design patterns in the first place and how adapting them to JavaScript will make your code more efficient and easier to work with.

The Flexibility of JavaScript

One of the most powerful features of the language is its flexibility. As a JavaScript programmer, you can make your programs as simple or as complex as you wish them to be. The language also allows several different programming styles. You can write your code in the functional style or in the slightly more complex object-oriented style. It also lets you write relatively complex programs without knowing anything at all about functional or object-oriented programming; you can be productive in this language just by writing simple functions. This may be one of the reasons that some people see JavaScript as a toy, but we see it as a good thing. It allows programmers to accomplish useful tasks with a very small, easy-to-learn subset of the language. It also means that JavaScript scales up as you become a more advanced programmer.

JavaScript allows you to emulate patterns and idioms found in other languages. It even creates a few of its own. It provides all the same object-oriented features as the more traditional server-side languages.

Let's take a quick look at a few different ways you can organize code to accomplish one task: starting and stopping an animation. It's OK if you don't understand these examples; all of the patterns and techniques we use here are explained throughout the book. For now, you can view this section as a practical example of the different ways a task can be accomplished in JavaScript.

If you're coming from a procedural background, you might just do the following:

```
/* Start and stop animations using functions. */
```

```
function startAnimation() {  
    ...  
}
```

```
function stopAnimation() {  
    ...  
}
```

This approach is very simple, but it doesn't allow you to create animation objects, which can store state and have methods that act only on this internal state. This next piece of code defines a class that lets you create such objects:

```
/* Anim class. */
```

```
var Anim = function() {  
    ...  
};  
Anim.prototype.start = function() {  
    ...  
};  
Anim.prototype.stop = function() {  
    ...  
};
```

```
/* Usage. */
```

```
var myAnim = new Anim();  
myAnim.start();  
...  
myAnim.stop();
```

This defines a new class called `Anim` and assigns two methods to the class's prototype property. We cover this technique in detail in Chapter 3. If you prefer to create classes encapsulated in one declaration, you might instead write the following:

```
/* Anim class, with a slightly different syntax for declaring methods. */
```

```
var Anim = function() {  
    ...  
};  
Anim.prototype = {  
    start: function() {  
        ...  
    },
```

```

    stop: function() {
        ...
    }
};

```

This may look a little more familiar to classical object-oriented programmers who are used to seeing a class declaration with the method declarations nested within it. If you've used this style before, you might want to give this next example a try. Again, don't worry if there are parts of the code you don't understand:

```
/* Add a method to the Function object that can be used to declare methods. */
```

```
Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
};
```

```
/* Anim class, with methods created using a convenience method. */
```

```
var Anim = function() {
    ...
};
Anim.method('start', function() {
    ...
});
Anim.method('stop', function() {
    ...
});
```

`Function.prototype.method` allows you to add new methods to classes. It takes two arguments. The first is a string to use as the name of the new method, and the second is a function that will be added under that name.

You can take this a step further by modifying `Function.prototype.method` to allow it to be chained. To do this, you simply return `this` after creating each method. We devote Chapter 6 to chaining:

```
/* This version allows the calls to be chained. */
```

```
Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
    return this;
};
```

```
/* Anim class, with methods created using a convenience method and chaining. */
```

```
var Anim = function() {
    ...
};
```

```
Anim.  
  method('start', function() {  
    ...  
  }).  
  method('stop', function() {  
    ...  
  });
```

You have just seen five different ways to accomplish the same task, each using a slightly different style. Depending on your background, you may find one more appealing than another. This is fine; JavaScript allows you to work in the style that is most appropriate for the project at hand. Each style has different characteristics with respect to code size, efficiency, and performance. We cover all of these styles in Part 1 of this book.

A Loosely Typed Language

In JavaScript, you do not declare a type when defining a variable. However, this does not mean that variables are not typed. Depending on what data it contains, a variable can have one of several types. There are three primitive types: booleans, numbers, and strings (JavaScript differs from most other mainstream languages in that it treats integers and floats as the same type). There are functions, which contain executable code. There are objects, which are composite datatypes (an array is a specialized object, which contains an ordered collection of values). Lastly, there are the `null` and `undefined` datatypes. Primitive datatypes are passed by value, while all other datatypes are passed by reference. This can cause some unexpected side effects if you aren't aware of it.

As in other loosely typed languages, a variable can change its type, depending on what value is assigned to it. The primitive datatypes can also be cast from one type to another. The `toString` method converts a number or boolean to a string. The `parseFloat` and `parseInt` functions convert strings to numbers. Double negation casts a string or a number to a boolean:

```
var bool = !!num;
```

Loosely typed variables provide a great deal of flexibility. Because JavaScript converts type as needed, for the most part, you won't have to worry about type errors.

Functions As First-Class Objects

In JavaScript, functions are first-class objects. They can be stored in variables, passed into other functions as arguments, passed out of functions as return values, and constructed at run-time. These features provide a great deal of flexibility and expressiveness when dealing with functions. As you will see throughout the book, these features are the foundation around which you will build a classically object-oriented framework.

You can create *anonymous functions*, which are functions created using the `function()` `{ ... }` syntax. They are not given names, but they can be assigned to variables. Here is an example of an anonymous function:

```
/* An anonymous function, executed immediately. */  
  
(function() {  
  var foo = 10;  
  var bar = 2;  
  alert(foo * bar);  
})();
```

This function is defined and executed without ever being assigned to a variable. The pair of parentheses at the end of the declaration execute the function immediately. They are empty here, but that doesn't have to be the case:

```
/* An anonymous function with arguments. */  
  
(function(foo, bar) {  
  alert(foo * bar);  
})(10, 2);
```

This anonymous function is equivalent to the first one. Instead of using `var` to declare the inner variables, you can pass them in as arguments. You can also return a value from this function. This value can be assigned to a variable:

```
/* An anonymous function that returns a value. */  
  
var baz = (function(foo, bar) {  
  return foo * bar;  
})(10, 2);  
  
// baz will equal 20.
```

The most interesting use of the anonymous function is to create a closure. A *closure* is a protected variable space, created by using nested functions. JavaScript has function-level scope. This means that a variable defined within a function is not accessible outside of it. JavaScript is also lexically scoped, which means that functions run in the scope they are defined in, not the scope they are executed in. These two facts can be combined to allow you to protect variables by wrapping them in an anonymous function. You can use this to create private variables for classes:

```
/* An anonymous function used as a closure. */  
  
var baz;  
  
(function() {  
  var foo = 10;  
  var bar = 2;  
  baz = function() {  
    return foo * bar;  
  };  
})();
```



```
baz(); // baz can access foo and bar, even though it is executed outside of the
      // anonymous function.
```

The variables `foo` and `bar` are defined only within the anonymous function. Because the function `baz` was defined within that closure, it will have access to those two variables, even after the closure has finished executing. This is a complex topic, and one that we touch upon throughout the book. We explain this technique in much greater detail in Chapter 3, when we discuss encapsulation.

The Mutability of Objects

In JavaScript, everything is an object (except for the three primitive datatypes, and even they are automatically wrapped with objects when needed). Furthermore, all objects are *mutable*. These two facts mean you can use some techniques that wouldn't be allowed in most other languages, such as giving attributes to functions:

```
function displayError(message) {
    displayError.numTimesExecuted++;
    alert(message);
};
displayError.numTimesExecuted = 0;
```

It also means you can modify classes after they have been defined and objects after they have been instantiated:

```
/* Class Person. */

function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype = {
    getName: function() {
        return this.name;
    },
    getAge: function() {
        return this.age;
    }
}

/* Instantiate the class. */

var alice = new Person('Alice', 93);
var bill = new Person('Bill', 30);

/* Modify the class. */
```

```
Person.prototype.getGreeting = function() {
  return 'Hi ' + this.getName() + '!';
};

/* Modify a specific instance. */

alice.displayGreeting = function() {
  alert(this.getGreeting());
}
```

In this example, the `getGreeting` method is added to the class after the two instances are created, but these two instances still get the method, due to the way the prototype object works. `alice` also gets the `displayGreeting` method, but no other instance does.

Related to object mutability is the concept of *introspection*. You can examine any object at run-time to see what attributes and methods it contains. You can also use this information to instantiate classes and execute methods dynamically, without knowing their names at development time (this is known as *reflection*). These are important techniques for dynamic scripting and are features that static languages (such as C++) lack.

Most of the techniques that we use in this book to emulate traditional object-oriented features rely on object mutability and reflection. It may be strange to see this if you are used to languages like C++ or Java, where an object can't be extended once it is instantiated and classes can't be modified after they are declared. In JavaScript, everything can be modified at run-time. This is an enormously powerful tool and allows you to do things that are not possible in those other languages. It does have a downside, though. It isn't possible to define a class with a particular set of methods and be sure that those methods are still intact later on. This is part of the reason why type checking is done so rarely in JavaScript. We cover this in Chapter 2 when we talk about duck typing and interface checking.

Inheritance

Inheritance is not as straightforward in JavaScript as in other object-oriented languages. JavaScript uses object-based (prototypal) inheritance; this can be used to emulate class-based (classical) inheritance. You can use either style in your code, and we cover both styles in this book. Often one of the two will better suit the particular task at hand. Each style also has different performance characteristics, which can be an important factor in deciding which to use. This is a complex topic, and we devote Chapter 4 to it.

Design Patterns in JavaScript

In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published a book titled *Design Patterns*. This book catalogs the different ways objects can interact with each other and it created a common vocabulary around the different types of objects. The blueprints for creating these different types of objects are called *design patterns*. The book describes these patterns in a somewhat language-agnostic way, so that they can be used anywhere. The book you are holding in your hands takes those patterns and applies them specifically to JavaScript.

The fact that JavaScript is so expressive allows you to be very creative in how design patterns are applied to your code. There are three main reasons why you would want to use design patterns in JavaScript:

1. *Maintainability*: Design patterns help to keep your modules more loosely coupled. This makes it easier to refactor your code and swap out different modules. It also makes it easier to work in large teams and to collaborate with other programmers.
2. *Communication*: Design patterns provide a common vocabulary for dealing with different types of objects. They give programmers shorthand for describing how their systems work. Instead of long explanations, you can just say, “It uses the factory pattern.” The fact that a particular pattern has a name means you can discuss it at a high level, without having to get into the details.
3. *Performance*: Some of the patterns we cover in this book are optimization patterns. They can drastically improve the speed at which your program runs and reduce the amount of code you need to transmit to the client. The flyweight (Chapter 13) and proxy (Chapter 14) patterns are the most important examples of this.

There are two reasons why you might *not* want to use design patterns:

1. *Complexity*: Maintainability often comes at a cost, and that cost is that your code may be more complex and less likely to be understood by novice programmers.
2. *Performance*: While some patterns improve performance, most of them add a slight performance overhead to your code. Depending on the specific demands of your project, this overhead may range from unnoticeable to completely unacceptable.

Implementing patterns is the easy part; knowing which one to use (and when) is the hard part. Applying design patterns to your code without knowing the specific reasons for doing so can be dangerous. Make an effort to ensure that the pattern you select is the most appropriate and won't degrade performance below acceptable limits.

Summary

The expressiveness of JavaScript provides an enormous amount of power. Even though the language lacks certain useful built-in features, its flexibility allows you to add them yourself. You can write code to accomplish a task in many different ways, depending on your background and personal preferences.

JavaScript is loosely typed; programmers do not declare a type when defining a variable. Functions are first-class objects and can be created dynamically, which allows you to create closures. All objects and classes are mutable and can be modified at run-time. There are two styles of inheritance you can use, prototypal and classical, and each has its own strengths and weaknesses.

Design patterns in JavaScript can be extremely helpful and beneficial, but they can also be detrimental if used improperly. In a language as lightweight as JavaScript, overly complex architectures can quickly bog down your application. Always make sure the style of programming you use and the patterns you select are right for the job.