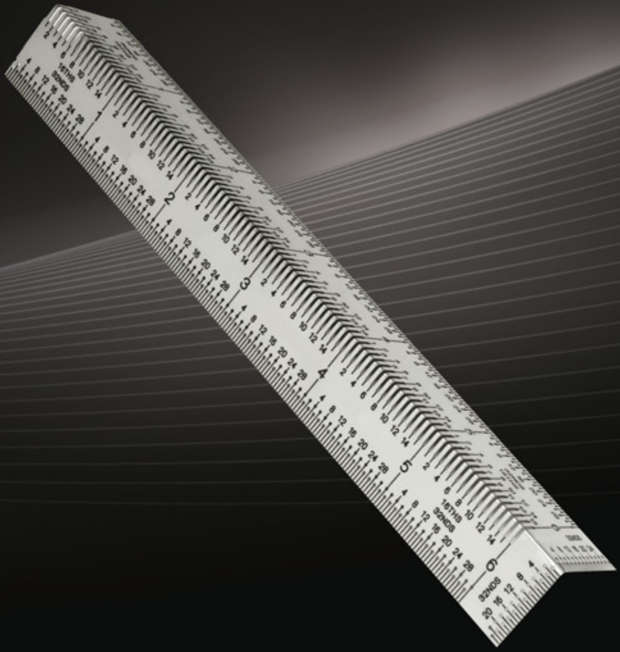
Microsoft

Microsoft®

ASP.NET and AJAX: Architecting Web Applications



Dino Esposito

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008940527

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 4 3 2 1 0 9

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ben Ryan
Developmental Editor: Lynn Finnel
Project Editor: Tracy Ball
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Kenn Scribner; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-28134

To the people who help me to smile and often smile, play and laugh with me.

—Dino

This page intentionally left blank

Contents at a Glance

Part I **The (Much Needed) Facelift for the Old Web**

1 Under the Umbrella of AJAX. 3

2 The Easy Way to AJAX 27

3 AJAX Architectures. 61

Part II **Power to the Client**

4 A Better and Richer JavaScript 101

5 JavaScript Libraries 129

6 AJAX Design Patterns. 163

7 Client-Side Data Binding 223

8 Rich Internet Applications 269

Index. 309

Table of Contents

Acknowledgments xi

Introduction xiii

Part I The (Much Needed) Facelift for the Old Web

1 Under the Umbrella of AJAX. 3

What Web Do We Want? 4

It's All About User Experience 4

Origins of the Web 7

Paradox of the Web 9

The Biggest Benefit of AJAX. 11

What's AJAX, Exactly? 12

The Paradigm Shift 14

AJAX and New Web Projects 17

Adding AJAX Capabilities 17

Architecture Is the Concern 18

The Case for Rich Internet Applications. 22

Summary 24

2 The Easy Way to AJAX 27

The ASP.NET AJAX Infrastructure 28

The Page's Script Manager. 28

The Microsoft JavaScript Library 35

Partial Rendering 37

The *UpdatePanel* Control 37

Programming Updatable Panels 43

Minimizing Data Transfer 47

Shades of Partial Rendering. 48

AJAX and JavaScript Injections. 53

Remote Methods 54

Widgets and Effects. 56

Summary 60

3 AJAX Architectures 61

The AJAX Service Layer Pattern 62

Architectural Overview 62

Inside the HTTP Façade 70

The AJAX Presentation Layer 79

Security Considerations 83

The AJAX Server Pages Pattern 87

Architectural Overview 88

The Classic Postback Model Revisited 90

Libraries in Action 92

Summary 97

Part II Power to the Client

4 A Better and Richer JavaScript 101

JavaScript Today 102

The Language and the Browser 102

Pillars of the Language 105

JavaScript (If Any) of the Future 108

The Microsoft AJAX Library 110

Overview of the Library 110

JavaScript Language Extensions 112

Object-Oriented Extensions 115

Framework Facilities 119

Summary 126

5 JavaScript Libraries 129

From Server Controls to JavaScript Widgets 130

The ASP.NET Factor 130

The Widget Factor 132

The jQuery Library 137

The Library at a Glance 138

The Core Library 140

jQuery Selectors 142

Working on Wrapped Sets 149

jQuery Utilities 151

Summary 161

6 AJAX Design Patterns. 163

Design Patterns and Code Development. 163

Generalities About Design Patterns 164

Patterns in AJAX Development 166

Patterns for JavaScript Development 168

The Singleton Pattern 169

The Model-View-Controller Pattern. 170

The On-Demand JavaScript Pattern. 175

The Predictive Fetch Pattern 178

Generalities of the Predictive Fetch Pattern 178

Creating a Reference Implementation 180

The Timeout Pattern 186

Generalities of the Timeout Pattern 187

A Timeout Pattern Reference Implementation. 188

Related Patterns. 192

The Progress Indicator Pattern 194

Generalities of the Progress Indicator Pattern 194

A Progress Indicator Reference Implementation 196

Canceling an Ongoing Remote Task. 206

Other Patterns 213

The Micro-Link Pattern. 213

The Cross-Domain Proxy Pattern 215

The Submission Throttling Pattern 218

Summary 221

7 Client-Side Data Binding 223

An Architectural Tour of ASP.NET Data Binding 224

Defining the HTML Template. 224

Defining the Data Source 230

Data Binding at the Time of AJAX 232

The Browser-Side Template Pattern 235

Generalities of the BST Pattern 235

Creating a BST Reference Implementation 238

The HTML Message Pattern 250

Generalities of the HM Pattern 250

Developing an HM Reference Implementation 253

A Look at ASP.NET AJAX 4.0. 260
ASP.NET AJAX Templates 260
ASP.NET Library for ADO.NET Data Services. 266
Summary 268

8 Rich Internet Applications 269
Looking for a Richer Web 269
The Dream of Binary Code Running over the Web 270
Browser Plug-ins 271
Microsoft Silverlight at a Glance 274
Elements of the Silverlight Architecture. 275
Graphics and Multimedia 277
Building Applications 279
The Programming Model of Microsoft Silverlight 282
WPF-Based User Interface 282
The .NET Base Class Library 286
Isolated Storage 289
Networking 295
Microsoft Silverlight and Code Security 302
The Security Model 302
Security Attributes 303
Secure by Design 306
Summary 308

Index 309

Acknowledgments

A team of people helped me to assemble this book.

Ben Ryan was sneakily convinced to support the project on a colorful Las Vegas night, during an ethnic dinner during which we watched waiters coming up from and going down to the wine cellar in transparent elevators.

Lynn Finnel just didn't want to let Dino walk alone in this key project after brilliantly coordinating at least five book projects in the past.

Kenn Scribner is now Dino's official book alter ego. Kenn started working with Dino on books back in 1998 in the age of COM and the Active Template Library. How is it possible that a book with Dino's name on the cover isn't reviewed and inspired (and fixed) by Kenn's unique and broad perspective on the world of software? The extent to which Kenn can be helpful is just beyond human imagination.

Roger LeBlanc joined the team to make sure that all these geeks sitting together at the same virtual desktop could still communicate using true English syntax and semantics.

I owe you all the (non-rhetorically) monumental "Thank you" for being so kind, patient, and accurate.

—Dino

This page intentionally left blank

Introduction

This book is the Web counterpart to another recently released book I co-authored with Andrea Saltarello: *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008). I wrote it, in part, in response to the many architectural questions—both small questions and big ones—that I was asked repeatedly while teaching ASP.NET, AJAX, and Silverlight classes.

Everybody in the industry is committed to AJAX. Everybody understands the impact of it. Everybody recognizes the enormous power that can be derived from its employment in real-world solutions.

Very few, though, know exactly how to make it happen. There are so many variations to AJAX and so many implementations that even after you have found one that suits your needs, you are left wondering whether that is the best possible option.

The fact is that AJAX triggered a chain reaction in the world of the Web. AJAX represents a change of paradigm for Web applications. And, as the history of science proves, a paradigm shift has always had a deep impact, especially in scenarios that were previously stable and consolidated.

I estimate that it will take about five years to absorb the word *AJAX* (and all of its background) into the new definition of the Web. And the clock started ticking about four years ago. The time at which we say "the Web" without feeling the need to specify whether it contains AJAX or not... well, that time is getting closer and closer. But it is not that time yet.

Tools and programming paradigms for AJAX, which were very blurry just a few years ago, are getting sharper every day. Whether we are talking about JavaScript libraries or suites of server controls, I feel that pragmatic architectures can be identified. You find them thoroughly discussed in Chapter 3, "AJAX Architectures."

Architecting a Web application today is mostly about deciding whether to prefer the *richness* of the solution over the *reach* of the solution. Silverlight and ASP.NET AJAX are the two platforms to choose from as long as you remain in the Microsoft ecosystem. But the rich vs. reach dilemma is a general one and transcends platforms and vendors. A neat answer to that dilemma puts you on the right track to developing your next-generation Web solution.

Who This Book Is For

I believe that this book is ideal reading for any professionals involved with the ASP.NET platform and who are willing or needing to find a solution that delivers a modern and rich user experience.

Companion Content

Examples of techniques and patterns discussed in the book can be found at the following site: *http://www.microsoft.com/learning/en/us/books/12926.aspx*.

Hardware and Software Requirements

You'll need the following hardware and software to work with the companion content included with this book:

- Nearly any version of Microsoft Windows, including Vista (Home Premium Edition, Business Edition, or Ultimate Edition), Windows Server 2003 and 2008, and Windows XP Pro.
- Microsoft Visual Studio 2008 Standard Edition, Visual Studio 2008 Enterprise Edition, or Microsoft Visual C# 2008 Express Edition, and Microsoft Visual Web Developer 2008 Express Edition.
- Microsoft SQL Server 2005 Express Edition, Service Pack 2 or Microsoft SQL Server 2005, Service Pack 3, or Microsoft SQL Server 2008.
- The Northwind database of Microsoft SQL Server 2000 is used to demonstrate data-access techniques. You can obtain the Northwind database from the Microsoft Download Center (*http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en*).
- 1.6 GHz Pentium III+ processor, or faster.
- 1 GB of available, physical RAM.
- Video (800 by 600 or higher resolution) monitor with at least 256 colors.
- CD-ROM or DVD-ROM drive.
- Microsoft mouse or compatible pointing device.

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *http://www.microsoft.com/learning/books/online/developer* and is updated periodically.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Microsoft ASP.NET and AJAX: Architecting Web Applications* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

This page intentionally left blank

Chapter 4
A Better and Richer JavaScript

Language is the source of misunderstandings.

—Antoine de Saint-Exupery

Aside from the social implications of it, the Web 2.0 from a technology viewpoint is mostly about running more JavaScript code on the client. You can't just take the standard JavaScript language that most browsers support today and ask any developer to write immensely capable applications using it. As a projectwide approach, it just doesn't scale and work the way you might expect. JavaScript is not like, say, C#. JavaScript is a very special type of language; it's probably not the language everybody would choose to use today to power up the client side of the Web. However, it's the only common language we have, and we have to stick to it to reach the largest audience.

So what if you want (or more likely need) more power on the client?

Be ready to write more JavaScript code; more importantly, be ready to import more JavaScript code written by others. Either of these two ways of using JavaScript is OK, as they are not mutually exclusive options.

JavaScript is not the perfect language, and, amazingly, it was not designed to be the super language to rule the Web. JavaScript is popular, and this is its major strength and most significant weakness. It's a strength because it allows you to reach virtually every browser and every user; it's a weakness because its widespread use makes implementing any important change or extension painful in terms of achieving compatibility.

In summary, I firmly believe that for the time being you can't just transform JavaScript into something else that is radically different from what the language is today. However, the Web has repeatedly proven to be a surprisingly dynamic and agile environment; so who really knows what could happen in five years? Giving a judgment today, I would say that a winning approach needs to evolve the language without breaking compatibility with all of today's browsers. It ultimately means creating new libraries that add new features to the language. However, these libraries must be created using the same core language and, ideally, they should also be stacked up and composed together in a recipe that suits any given application.

In this chapter and the next, I'll review two JavaScript libraries that work well together today and that will probably evolve together in the near future: the Microsoft AJAX library and the jQuery library.

JavaScript Today

AJAX would not be possible without JavaScript. And this happens not because JavaScript is such a powerful language, but because JavaScript is so popular and built in nearly the same form in virtually all browsers released in the past five years.

Three ingredients, combined in the right doses, almost spontaneously originated the AJAX paradigm shift: a standard browser-hosted programming language (JavaScript), a standard object model to fully represent the document being viewed (the W3C's Document Object Model), and a sufficiently rich browser object model that includes the key *XMLHttpRequest* object.

Separating these elements is almost impossible nowadays. JavaScript is more than a simple programming language and, as you'll see later in the chapter, modern libraries reflect that.

The Language and the Browser

JavaScript is a language tailor-made for the Web and, more specifically, for the browser. In fact, there's no compiler currently available that allows you to create binaries from a bunch of JavaScript source files.

The only exception I'm aware of is the Managed JScript compiler for the .NET Framework. However, I don't recall ever meeting someone who used it concretely to build applications and not simply as a proof of some concepts.

I won't stray too far from the truth by saying that there's no life for JavaScript outside the realm of a Web browser. Of course, this is largely due to where JavaScript originated and the purpose it fulfilled at the time. Let's briefly recall the origins of the language.

Original Goals of the Language

The first appearance of JavaScript as a browser-hosted language dates back to late 1995, when the first beta of Netscape Navigator 2 was released. JavaScript was introduced to give authors of Web documents the ability to incorporate some logic and action in HTML pages. Before then, a Web page was essentially a static collection of HTML tags and text. Historically, the first significant enhancement made to the syntax of HTML was the support for tags to include script code.

JavaScript was not designed to be a classic and cutting-edge programming language—not even by the standards of 15 years ago. The primary goal of its designers was to create a language that resembled a simpler Java that could be used with ease by nonexpert page authors.

To some extent, the design of JavaScript was influenced by many languages, but the predominant factor was simplicity. It was named JavaScript because the language was essentially meant to be a powerful language (like Java) but focused on scripting. No other relationships, beyond the deliberate reference in the name, exist between Java and JavaScript.

As a result, JavaScript is an interpreted and weakly typed language that also supports dynamic binding and objects. JavaScript, however, is not a fully object-oriented language.

Note Originally developed at Netscape by Brendan Eich, JavaScript was first named *LiveScript*. The name was changed to JavaScript when Netscape added support for Java technology in its Navigator browser. The *script* suffix was simply meant to be the script version of an excellent programming language like Java. In no way was the language supposed to be a spinoff of Java.

Later, Microsoft created a similar language for its Internet Explorer browser and named it JScript to avoid trademark issues. In 1997, JavaScript was submitted to the European Computer Manufacturers Association (ECMA) International for standardization. The process culminated a couple of years later in the standardized version of the language named ECMAScript.

The Scripting Engine

Being an interpreted language, JavaScript requires an ad hoc run-time environment to produce visible effects from the source code. The run-time environment is often referred to as the browser's *scripting engine*. As such, the JavaScript's run-time environment can be slightly different from one browser to the next. The result is that the same JavaScript language feature might provide a different performance on different browsers and might be flawed on one browser while working efficiently on another one.

This fact makes it hard to write good, cross-browser JavaScript code and justifies the love/hate relationship (well, mostly hate) that many developers have developed with the language over the years.

The diagram in Figure 4-1 shows the overall structure of a scripting engine.

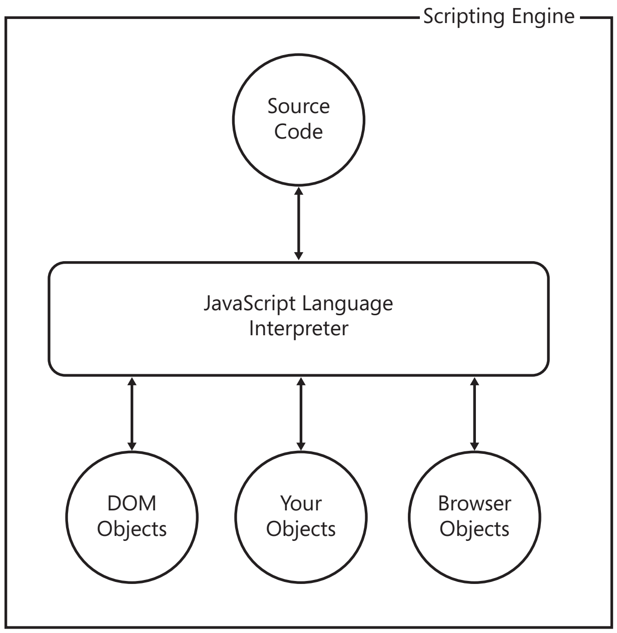


FIGURE 4-1 The browser's scripting engine

The engine is a component that is hosted in the browser and receives the source code to process. Armed with language knowledge, the engine can resolve any name in the source code that can be mapped to a syntax element—keywords, variables, local functions, and objects.

In addition, the source code processed within a Web browser is likely populated with specific objects coming from a variety of sources. For example, you can find DOM objects to access the content being displayed in the page as well as browser-specific objects such as *XMLHttpRequest* and *window*. Furthermore, any libraries you reference from the page are also published to the engine. After the script has been loaded, the browser runs the script through the engine. This action results in the functionality defined by the commands in the code.

As mentioned, although JavaScript is definitely a stable language that hasn't faced significant changes for 10 years now, virtually any broadly used library is packed with forks in code to distinguish the behavior of different browsers and ensure the same overall interface.

One of the first rules—if not the first rule—you should follow to write good AJAX applications is get yourself a powerful JavaScript library that adds abstraction and features to the JavaScript language and that works in a cross-browser manner.

Note As far as the ASP.NET platform is concerned, the good news is that you have neither to reinvent the wheel nor to invent your own wheel to proceed. In fact, the AJAX extensions to ASP.NET include a cross-browser core library that you can use as the foundation for any JavaScript code you might need beyond ready-made objects and functionalities.

Recognized Flaws

As you'll see in a moment, JavaScript has a number of drawbacks, both technical and infrastructural. In spite of all these factors, though, JavaScript works just great for the majority of Web applications. And nothing any better has been invented yet.

All things considered, the limitations of JavaScript can be summarized as two elements: it is an interpreted language, and it is not fully object oriented. The former drawback makes the language significantly slower than a compiled language. The latter makes it harder for developers to write complex code.

These were not limitations in the beginning, about 10 years ago. Nonetheless, they are now limitations that become more evident every day. Replacing JavaScript, however, is not something that can happen overnight.

JavaScript is so popular and widely used that making any breaking changes to it would break too many applications. Yet the direction that JavaScript is taking in light of AJAX addresses the two aforementioned limitations.

The Google Chrome browser (which you can read more about at *http://www.google.com/chrome*) comes with an open-source JavaScript engine that compiles source code to native machine code before executing it. As a result, Chrome runs JavaScript applications at the speed of a compiled binary, which is significantly better than any bytecode or interpreted code.

The Microsoft AJAX library, as well as other popular JavaScript libraries, such as Prototype, offers some built-in features to add inheritance to JavaScript objects and flavors of object orientation.

Note Chrome and its V8 JavaScript engine are taking an innovative approach to dealing with the growth in size and complexity of JavaScript code in AJAX applications. Other libraries are trying to offer more powerful instruments to raise the abstraction level of the original JavaScript language. We are not seeing either a brand new language or an improved core language, but something is happening on the client side to make JavaScript code more effective.

Pillars of the Language

In more than 10 years of existence, JavaScript has never been as central a technology in the world of Web computing as it is today following the arrival of AJAX. JavaScript code in the average Web page has grown from just a few lines of trivial code that just scripts page elements to hundreds of kilobytes of code providing rich object models, if not true frameworks.

Because it was not created to be a spinoff of a true compiled programming language, JavaScript supports all the syntax elements of a common structured programming language, such as *if*, *switch*, *for*, and *while* statements. Types are not strongly enforced and are associated with values rather than with variables. Let's briefly review the pillars of the JavaScript language.

Note Any piece of source code written in JavaScript and completely delivered to a browser is immediately executable. Clearly, this provides the potential for malicious code to be downloaded and run on the client computer. To contain the risk, the browser runs any script within a *sandbox*. A sandbox is a virtual environment where hosted programs can perform only controlled actions and are typically not granted permissions to operate on the file system and the local hardware.

In addition, browsers also commonly restrict scripts from accessing any information from an external site. This is known as a *same origin policy*. Violating the same origin policy may result in a cross-site scripting attack.

Objects as Dictionaries

The JavaScript language allows you to use objects, but it doesn't natively support all principles of object-oriented programming (OOP), such as inheritance, encapsulation, and polymorphism. To some extent, some of these principles can be recognized here and there in the language's capabilities; however, JavaScript can't be described as a fully object-oriented (OO) language.

The primary reason for not cataloging JavaScript as an OO language is that the definition of an *object* you get from JavaScript is different from the commonly accepted idea of an object you get out of classic OO languages such as C++ or C#.

In C# and C++, you create new objects by telling the runtime which *class* you want to instantiate. A *class* is a fixed template used for the object creation. A class defines the properties and methods an object will have, and these properties and methods are forever fixed. In C# and C++, you can't manipulate the structure of an object by adding or removing properties and methods at runtime.

In JavaScript, objects are essentially dictionaries of values or associative arrays. An object is a container of name/value pairs that can be added at any time, and especially at runtime. In an attempt to express a JavaScript object via a C# notation, you would probably resort to something similar to the following:

```
Dictionary<string, object>
```

The property name is a string that acts as the key in the dictionary, as shown here:

```
var obj = new YourJavaScriptObject();
obj["Property"] = "Hello";
```

You can also use an alternate syntax based on the *dot* notation. The effect is the same:

```
obj.Property = "Hello";
```

JavaScript objects contain more than just a dictionary of values. In particular, they contain the *prototype* object. The prototype is like a directory that defines the public interface of the object. By acting on the prototype, you can augment the capabilities of the object in a fully dynamic manner.

Functions as Objects

Another fundamental characteristic of JavaScript is that functions are first-class language elements and objects themselves. In other words, functions might have properties and can be passed around and interacted with as you would do with any other object.

You can use the *new* operator with a function. When you do so, you get an entirely new object and can reference it internally using the *this* keyword. Just like any other object, the function has its own *prototype* property that determines the public interface of the new object:

```
MyPet = function (name, isDog)
{
   this._name = name;
   this._isDog = isDog;
}
MyPet.prototype = {
   get_Name = function() {return this._name;},
   get_IsDog = function()  {return this._isDog;}
}
```

Given the preceding code, you can use the *new* operator on the *MyPet* function and invoke the members in the prototype.

Dynamic Typing

The nature of objects and functions makes JavaScript a very dynamic language. Types are not an exception and don't force developers to follow strict rules as in a classic programming language.

Like many other scripting languages, JavaScript recognizes a few primitive types (string, number, date, Boolean) but doesn't let you declare a variable of a given, fixed type. Variables are untyped on declaration and can hold values of different types during their lifetime. As mentioned, in JavaScript types are associated with values rather than with variables.

```
x = "1";   // It is a string
x = 1;     // It is a number
```

For this reason, equality operators work in a slightly different manner. Given the following lines of code, what would be the result of the expression *x == y*?

```
x = "1";
y = 1;
```

If you look at the code from an OO perspective, you can have only one answer: *false*. Quite surprisingly, instead, in JavaScript *x==y* returns *true* because the comparison is made on the value, not the type. To get the expected result, you must switch to the === operator, which checks value *and* type.

JavaScript provides the *typeof* built-in function to test the type of an object. Another approach is *duck typing*. Duck typing basically consists of providing the freedom of invoking on an object any methods it *seems* to have. If it does not have a particular method, you just get a run-time exception. Duck typing originates from the statement: *If it walks like a duck and quacks like a duck, I would call it a duck*.

Closures and Prototypes

The three pillars of object orientation can be implemented in JavaScript to various degrees. For example, encapsulation is easy to get via the *var* keyword in a *closure* model. Encapsulation is impossible to achieve if you are working with a *prototype* model. The prototype model makes it easy to build inheritance, and polymorphism can be obtained via a combination of functions and duck typing.

There are two main models for designing classes in JavaScript: closures and prototypes. The models are not entirely equivalent, so choosing one over the other is a matter of evaluating the tradeoffs. Also, the performance you get for both models in the major browsers is not the same. Let's learn more about the closure model.

In the closure model, a custom object is a single function where all members are defined together within the same (closed) context, as shown here:

```
// The Person object is entirely defined here
Person = function()
{
    var _firstName;   // private member
    var _lastName;    // private member
    this.get_FirstName = function() { return this._firstName; }
    this.get_LastName = function() { return this._lastName; }
}
```

The use of the *var* keyword keeps a member declaration local to the context and ensures data encapsulation. Accessing *_firstName* and *_lastName* members from outside the closure is impossible, as is the case when accessing a private member from outside a class definition in C# or C++. Members not tagged with the *var* keyword are meant to be public. The object declaration occurs in a single place and through a unique constructor. Using objects built as closures can be memory intensive because a new instance is required for any work—just like in C# or C++.

The prototype model defines the public structure of the class through the built-in *prototype* object. The definition of an object, however, is not centered around a single point of scope. Here's how the object definition changes if you opt for the prototype model:

```
Person = function (firstName, lastName)
{
   this._firstName = firstName;
   this._lastName = lastName;
}
Person.prototype = {
   get_FirstName = function() {return this._firstName;},
   get_LastName = function()  {return this._lastName;}
}
```

As you can see, the object constructor and members are clearly separated. Members are shared by all instances and are private only by convention. Using the *var* keyword in the definition of, say, *_firstName* would make it private and inaccessible. On the other hand, not using the *var* keyword keeps the member public and therefore visible from the outside.

Because members of the *prototype* are global and static, the prototype model reduces the amount of memory required by each instance of the object and makes object instantiation a bit faster.

Note Prototypes have a good load time in nearly all modern-day browsers, and load times are excellent in Firefox. On the other hand, closures are faster than prototypes in all recent versions of Internet Explorer.

JavaScript (If Any) of the Future

Two pillars carry the whole weight of the Web: HTML and JavaScript. Neither of them seems to be entirely appropriate in the age of AJAX. And neither can be blissfully dismissed and replaced for compatibility and interoperability reasons. Regarding JavaScript, what can we do?

Like HTML, JavaScript is very efficient in doing the few and relatively simple things it was originally designed to do. The point is that the community of developers needs much more—more programming power and more performance.

Personally, I value programming power and language expressivity more than performance. To some extent, performance and JavaScript still sound to me like incompatible concepts.

Performance is especially relevant in a scenario where an ounce of performance lost in some task might be automatically multiplied by some factor, such as the growing number of requests. With JavaScript, frankly, there are no such risks. The JavaScript code runs on the client and on a computer that serves a single user at a time. There's no bad performance multiplier around.

JavaScript performance can become an issue—but not really a showstopper—only when you have so many lines of code (something like several hundred kilobytes) that it just takes too much to produce a user-friendly result.

Improving JavaScript might be desirable. But if so, how should that be done? There are two main schools of thought, plus a clever ploy.

Overhauling the Language

The specification for JavaScript 2.0 is currently being discussed and defined. You can find more details at *http://www.mozilla.org/js/language/evolvingJS.pdf*. JavaScript 2.0 is expected to be a significant overhaul of the language.

The most radical change that will come with JavaScript 2.0 is support for real classes and interfaces. The following syntax should be acceptable in the next version:

```
class Person
{
  this.FirstName = "dino";
  this.LastName = "esposito";
}
var p = new Person();
```

Compile-time type checking is another aspect waiting for improvement. A component that requires *strict* mode will have static type checking and a number of other checks performed before execution, such as verification that all referenced names are known and that only comparisons between valid types are made.

Namespaces and packages complete the set of hot features slated for the next JavaScript. A package is a library of code that is automatically loaded only on demand.

It's All About Security

Another camp sees the future of JavaScript in a different manner. This camp is well represented by Douglas Crockford—one of the creators of JSON. According to Douglas, security is the biggest concern for JavaScript developers. So by simply making JavaScript a more secure programming environment, we would make JavaScript a better environment.

Douglas suggests adding a *verifier* to analyze the source and spot unsafe code and a *transformer* to add indirection and run-time checks around critical instructions. More in general, the vision put forth by Douglas is centered on the idea of improving the language by making today's de facto standard solutions a native part of the language.

Google's V8 Engine

As mentioned, a new approach to JavaScript programming is coming out with Google's Chrome browser—the V8 engine. V8 is a new JavaScript engine specifically designed for optimized execution of *large* JavaScript applications.

The basic idea is that the browser operates as a just-in-time JavaScript compiler, wrapping functions into memory objects and turning them into machine code. In addition to dynamic machine-code generation, the increment of improved performance is the result of a couple of other factors: fast property access and efficient garbage collection. For more information, check out *http://code.google.com/p/v8*.

The Microsoft AJAX Library

A truly powerful JavaScript library today can't ignore the dependencies existing between the language itself and the Document Object Model (DOM) and Browser Object Model (BOM). Subsequently, a modern JavaScript library is made of three fundamental pieces: a flavor of object orientation, facilities for visual effects, and a network stack.

It is not coincidental that this is also the recipe for the Microsoft AJAX library—one of the pillars of the Microsoft strategy for AJAX. Initially developed to back up the ASP.NET AJAX Extensions 1.0, and successively integrated in ASP.NET 3.5, the library is still being improved and enhanced for ASP.NET 4.0.

The next release of ASP.NET is expected to ship a stronger and more powerful client platform that results from the integration of the newer AJAX library and the newest version of another quite popular and largely complementary library—the jQuery library.

Overview of the Library

The Microsoft AJAX library is written in JavaScript, although with a strong sense of object orientation. ASP.NET AJAX takes the JavaScript language to the next level by adding some type-system extensions, the notion of namespace and interface, plus facilities for inheritance. In addition, the ASP.NET AJAX JavaScript supports enumerations and reflection, and it has a number of helper functions to manipulate strings and arrays.

Constituent Files

The Microsoft AJAX library is coded using the base set of instructions that characterize the core JavaScript language, and it is persisted to a set of *.js* files. These *.js* files are not installed as distinct files on the Web server when you install ASP.NET. They are embedded as resources into the ASP.NET AJAX assembly—*system.web.extensions*. If you want them available as distinct files (for example, for your home perusal), go to *http://msdn2.microsoft.com/en-us/asp.net/bb944808.aspx*, check the license agreement, and get them as a single downloaded compressed file.

We already hinted at it in Chapter 2, "The Easy Way to AJAX," but let's briefly review in Table 4-1 the files that make up the library.

TABLE 4-1 Files That Form the Microsoft AJAX Library

Script	Description
MicrosoftAjax.js	A core part of the library, this file contains object-oriented extensions, the network stack, and a number of facilities, such as those for tracing and debugging.
MicrosoftAjaxWebForms.js	This file contains script functions to support ASP.NET partial rendering. In particular, it defines the client-side engine and programming interface for partial rendering.
MicrosoftAjaxTimer.js	This file contains the client-side programming interface of the *Timer* server control, a built-in control that comes with ASP.NET AJAX. The control creates a timer on the client and makes it post back upon timeout.

As you can see, these are plain JavaScript files that can be linked from any sort of Web page regardless of the technology it is written for—PHP, classic ASP.NET, ASP, or even plain HTML.

Linking the Microsoft AJAX Library

In ASP.NET 3.5 pages, you don't need to load files from the Microsoft AJAX library explicitly. This is a viable option when you don't have a customized version of the files to load. If you embed a *ScriptManager* control in your pages, the control will automatically recognize the Microsoft AJAX library files you need and will download them as required.

By default, script files will be extracted from the resources of the *system.web.extensions* assembly. If you hold your own copies of the scripts and want to reference them instead, you use the *ScriptManager* control as shown here:

```
<asp:ScriptReference Name="MicrosoftAjax.js"
                     Path="./MyScripts/MicrosoftAjax.js" />
```

You need the *Name* property to identify the name of the embedded resource that contains the client script file. The *Path* property can optionally be used to specify the physical server location where the named script file has to be loaded from.

When both *Name* and *Path* are specified, *Path* is the winner. Does it really make sense to specify both? Sure it does. When both properties are specified, you actually *replace* the standard *MicrosoftAjax.js* with the specified script.

Tip This trick can be used to take advantage of the script-related services of the *ScriptManager* control and also in scenarios where your pages are not dependent on the Microsoft AJAX library. By setting the *Name* property to *MicrosoftAjax.js* and the *Path* property to, say, *jquery.js*, you load *jQuery* instead of *Microsoft AJAX* while taking advantage of all the extra facilities of the *ScriptManager* control that we reviewed in Chapter 2. Read the full story at *http://weblogs.asp.net/bleroy/archive/2008/07/07/using-scriptmanager-with-other-frameworks.aspx*.

Note As general advice, I suggest that to reference a script file you don't strictly need the *ScriptManager* control. However, you should always consider using the *ScriptManager* control because of the handy services it provides, such as its ability to detect script duplicates and its free compression and localization.

No Bells and Whistles

As you'll see in greater detail in a moment, the Microsoft AJAX library provides core JavaScript services such as type extensions, OOP flavors, and an AJAX-enabled network stack. It doesn't provide any facilities for adding visual effects to your pages.

The integration between Microsoft AJAX library and jQuery that is coming out with the next version of ASP.NET will make up for this. You'll have a script framework that offers a richer JavaScript with advanced and commonly used widgets such as those provided by jQuery.

Let's dig out now the key capabilities of the Microsoft AJAX library.

JavaScript Language Extensions

The JavaScript language features a set of built-in objects, including *Function*, *Object*, *Boolean*, *Array*, *Number*, and *String*. All intrinsic objects have a read-only property named *prototype*. The *prototype* property provides a base set of functionality shared by any new instance of an object of that class.

New functionality can be added to each object by extending and improving its prototype. This is exactly what the Microsoft AJAX library does.

Primitive Types

The Microsoft AJAX library contains code that defines new objects and extends existing JavaScript objects with additional functionality. Table 4-2 lists the main global objects defined in the library and explains how they relate to original JavaScript types.

TABLE 4-2 Top-Level Objects in the Microsoft AJAX Library

Object	Description
Array	Extends the native *Array* object. This object groups static methods to add, insert, remove, and clear elements of an array. It also includes static methods to enumerate elements and check whether a given element is contained in the array.
Boolean	Extends the native *Boolean* object. This object defines a static *parse* method to infer a *Boolean* value from a string or any expression that evaluates to a *Boolean* value.
Date	Extends the native *Date* object with a couple of instance methods: *localeFormat* and *format*. These methods format the date using the locale or invariant culture information.

TABLE 4-2 Top-Level Objects in the Microsoft AJAX Library

Object	Description
Error	Defines a static create method to wrap the JavaScript *Error* object and add a richer constructor to it. This object incorporates a couple of properties—*message* and *name*—to provide a description of the error that occurred and identify the error by name. A number of built-in error objects are used to simulate exceptions. In this case, the *name* property indicates the name of the exception caught.
Function	Extends the native *Function* object. This object groups methods to define classes, namespaces, delegates, and a bunch of other object-oriented facilities.
Number	Extends the native *Number* object. This object defines a static *parse* method to infer a numeric value from a string or any expression that evaluates to a numeric value. In addition, it supports a pair of static formatting methods: *localeFormat* and *format*.
Object	Extends the native *Object* object. This object groups methods to read type information, such as the type of the object being used.
RegExp	Wraps the native *RegExp* object.
String	Extends the native *String* object. This object groups string manipulation methods, such as trim methods and *endsWith* and *startsWith* methods. In addition, it defines static *localeFormat* and *format* methods that are close relatives of the *String.Format* method of the managed *String* type.

After the Microsoft AJAX library has been added to an application, the following code will work just fine:

```
var s = "Dino";
alert(s.startsWith('D'));
```

The native JavaScript *String* object doesn't feature either a *startsWith* or an *endsWith* method; the extended AJAX *String* object, instead, does.

New Types

As mentioned, it's only in a future version of JavaScript that you can start creating new complex and custom types as you do today in classic object-oriented languages. The Microsoft AJAX library, though, provides its own application programming interface (API) to let you register new objects—essentially custom JavaScript functions—with the library and use them as classes with an object-oriented flavor.

No new keyword is added for compatibility reasons, but a couple of new methods must be used to wrap the definition of a new type, as shown next for the sample *MyClass* type:

```
Type.registerNamespace("Samples");
Samples.MyClass = function ()
{
  ⋮
}

// Other blocks of code here for class members
⋮
Samples.MyClass.registerClass("Samples.MyClass");
```

Enumerations are a special breed of a new type in JavaScript. As in the .NET Framework, an enumeration represents an easily readable alternative to integers. Here's a sample definition for an enumerated type in JavaScript:

```
Type.registerNamespace("Samples");

// Define an enumeration type and register it.
Samples.Color = function() {};
Samples.Color.prototype =
{
    Red:    0xFF0000,
    Blue:   0x0000FF,
    Green:  0x00FF00,
    White:  0xFFFFFF
}
Samples.Color.registerEnum("Samples.Color");
```

To register an enumerated type, you use a tailor-made registration function—the *registerEnum* function.

Shorthand Functions

I would find it hard to believe that most of you reading this book have never made the mistake of using the name of the HTML element in a page as a shortcut to get the corresponding DOM reference. Suppose you have a text box element named *TextBox1* in the client page. The following script code won't work on all browsers:

```
alert(TextBox1.value);
```

The correct form ratified by the World Wide Web Consortium (W3C) paper for the HTML DOM standard is shown here:

```
alert(document.getElementById("TextBox1").value);
```

The correct form is clearly more verbose and bothersome to write over and over again. The Microsoft AJAX library comes to the rescue with the *$get* global function. Simply put, the *$get* function is a shortcut for the *document.getElementById* function. If the Microsoft AJAX library is in use, the following expression is fully equivalent to the one just shown:

```
alert($get("TextBox1").value);
```

The *$get* function has two overloads. If you call *$get* passing the sole ID, the function falls back into *document.getElementById*. Alternatively, you can specify a container as the second argument, as shown here:

```
var parent = $get("Div1");
$get("TextBox1", parent);
```

If the container element supports the *getElementById* method, the function returns the output of *element.getElementById*; otherwise, the *$get* function uses the DOM interface to explore the contents of the subtree rooted in the element to locate any node with the given ID.

Although *$get* is only an alias for a regular JavaScript function, it is often mistaken for a new language element. Other similar shortcuts exist in the library to create objects and add or remove event handlers.

Note The *$get* function has a lot in common with jQuery's *$* root object. To be precise, early builds of the Microsoft AJAX library were still using the same *$* expression that was renamed later to avoid collisions. The *$get* object in the Microsoft AJAX library is merely a direct DOM selector that just filters by ID. The *$* object in jQuery, instead, is a full selector that supports a much richer CSS-based syntax to filter DOM elements to return.

Object-Oriented Extensions

In JavaScript, the *Function* object is the main tool you use to combine code with properties and forge new components. In the Microsoft AJAX library, the *Function* object is extended to incorporate type information, as well as namespaces, inheritance, interfaces, and enumerations.

Namespaces and Classes

A namespace provides a way of grouping and classifying the types belonging to a library. Not a type itself, a namespace adds more information to the definition of each type in the library to better qualify it.

All custom JavaScript functions belong to the global space of names. In the Microsoft AJAX library, you can associate a custom function with a particular namespace, for purely organizational reasons. When declaring a custom type in the Microsoft AJAX library, you can do as follows:

```
Type.registerNamespace("Samples");
Samples.Person = function Samples$Person(firstName, lastName)
{
    this._firstName = firstName;
    this._lastName = lastName;
}

// Define the body of all members
function Samples$Person$ToString()
{
   return this._lastName + ", " + this._firstName;
}
:
:

// Define the prototype of the class
Samples.Person.prototype =
{
    ToString:      Samples$Person$ToString,
    get_FirstName: Samples$Person$get_FirstName,
    set_FirstName: Samples$Person$set_FirstName,
    get_LastName:  Samples$Person$get_LastName,
    set_LastName:  Samples$Person$set_LastName
}

// Register the class
Samples.Person.registerClass("Samples.Person");
```

The *Type.registerNamespace* method adds the specified namespace to the run-time environment. In a way, the *registerNamespace* method is equivalent to using the *namespace {…}* construct in C#. The *Samples.Person* function defined following the namespace declaration describes a *Person* type in the *Samples* namespace. Finally, the newly defined function must be registered as a class with the Microsoft AJAX library framework. You use the *registerClass* method on the current function.

The *registerClass* method takes a number of parameters. The first parameter is mandatory, and it indicates the public name that will be used to expose the JavaScript function as a class. Additional and optional parameters (not shown in the preceding code) are the parent class, if there is any, and any interface implemented by the class. We'll get into this in just a moment.

The Microsoft AJAX library follows the prototype model (as opposed to closures) to define its own custom types. The goal of the ASP.NET AJAX team was to deliver a model that provided the best quality and performance on the largest number of browsers. Prototypes have a good load time in all browsers; and indeed, they have excellent performance in Firefox. Furthermore, prototypes lend themselves well, more than closures do, to debugging as far as object instantiation and access to private members are concerned.

Note In the definition of a new class, you can use an anonymous function or a named function. In terms of syntax, both solutions are acceptable. The convention, though, is that you opt for named functions and name each function after its fully qualified name, replacing the dot symbol (.) with a dollar symbol ($). The convention is justified by the help this approach provides to IntelliSense in Microsoft Visual Studio 2008.

Inheritance and Polymorphism

Let's now define a new class, *Citizen*, that extends *Person* by adding a new property: a national identification number. Here's the skeleton of the code you need:

```
// Declare the class
Samples.Citizen = function Samples$Citizen(firstName, lastName, id)
{
   ⋮
}

// Define the prototype of the class
Samples.Citizen.prototype =
{
   ⋮
}
```

```
// Register the class
Samples.Citizen.registerClass("Samples.Citizen", Samples.Person);
```

Note that the first argument of *registerClass* is a string, but the second one has to be an object reference. The second argument indicates the object acting as the parent of the newly created object. Let's flesh out this code a bit.

In the constructor, you'll set some private members and call the base constructor to initialize the members defined on the base class. The *initializeBase* method (defined on the revisited *Function* object you get from the library) retrieves and invokes the base constructor:

```
Samples.Citizen = function Samples$Citizen(firstName, lastName, id)
{
    Samples.Citizen.initializeBase(this, [firstName, lastName]);
    this._id = id;
}
```

You pass *initializeBase* the reference to the current object as well as an array with any parameters that the constructor to call requires. You can use the *[...]* notation to define an array inline. If you omit the *[...]* notation, be ready to handle a parameter count exception.

Quite often, developers derive a class because they need to add new members or alter the behavior of an existing method or property. Object-oriented languages define a proper keyword to flag members as overridable. How is that possible in JavaScript?

Any member listed in the prototype of an object is automatically public and overridable. Here's the prototype of the *Citizen* class:

```
Samples.Citizen.prototype =
{
    ToString:    Samples$Citizen$ToString,
    get_ID:      Samples$Citizen$get_ID
}
```

The class has a read-only *ID* property and overrides the *ToString* method defined in the parent class. Let's have a look at the implementation of the overriding method:

```
function Samples$Citizen$ToString()
{
    var temp = Samples.Citizen.callBaseMethod(this, 'ToString');
    temp += "  [" + this._id + "]";
    return temp;
}
```

You use *callBaseMethod* to invoke the same method on the parent class. Defined on the *Function* class, the *callBaseMethod* method takes up to three parameters: the instance, the name of the method, plus an optional array of arguments for the base method.

As mentioned earlier, the *ToString* method on the *Person* class returns a *LastName, FirstName* string. The *ToString* method on the *Citizen* class returns a string in the following format: *LastName, FirstName [ID]*.

Note When the prototype model is used, JavaScript has no notion of private members because no common closure can be provided for all methods contributing to the same object. As a result, private members are conventionally indicated by the underscore symbol (_) prefixing their names. They're still public and accessible, though.

Interfaces

An interface describes a group of related behaviors that are typical of a variety of classes. In general, an interface can include methods, properties, and events; in JavaScript, it contains only methods.

Keeping in mind the constraints of the JavaScript language, to define an interface you create a regular class with a constructor and a prototype. The constructor and each prototyped method, though, will just throw a not-implemented exception. Here's the code for the sample *Sys.IDisposable* built-in interface:

```
Type.registerNamespace("Sys");
Sys.IDisposable = function Sys$IDisposable()
{
    throw Error.notImplemented();
}
function Sys$IDisposable$dispose()
{
    throw Error.notImplemented();
}
Sys.IDisposable.prototype =
{
    dispose: Sys$IDisposable$dispose
}
Sys.IDisposable.registerInterface('Sys.IDisposable');
```

The following statement registers the *Citizen* class, makes it derive from *Person*, and implements the *IDisposable* interface:

```
Samples.Citizen.registerClass('Samples.Citizen',
           Samples.Person, Sys.IDisposable);
```

To implement a given interface, a JavaScript class simply provides all methods in the interface and lists the interface while registering the class:

```
function Samples$Citizen$dispose
{
   this._id = "";
}

Samples.Citizen.prototype =
{
   dispose: Samples$Citizen$dispose
   ⋮
}
```

Note, though, that you won't receive any run-time error if the class that claims to implement a given interface doesn't really support all the methods. You will receive an error if a caller happens to invoke an interface function your class didn't implement, so by convention all interface methods should be implemented.

If a class implements multiple interfaces, you simply list all required interfaces in the *registerClass* method as additional parameters. Here's an example:

```
Sys.Component.registerClass('Sys.Component', null,
      Sys.IDisposable,
      Sys.INotifyPropertyChange,
      Sys.INotifyDisposing);
```

As you can see, in this case you don't have to group interfaces in an array.

Framework Facilities

Many layers of code form the Microsoft AJAX library, including a layer specifically created to smooth the creation of rich UI controls with AJAX capabilities. (See *http://www.codeplex.com/AjaxControlToolkit* for example controls.) This particular aspect of the library, though, is expected to evolve significantly in the next release of ASP.NET.

Let's focus instead on other core facilities you find in the library, such as event handling, debugging, and networking. To start out, let's attack with reflection capabilities.

Reflection

While debugging some JavaScript code, isn't it a bit frustrating when you need to know the actual type of a variable and cannot get it exact? In general, *reflection* refers to the ability of a function to examine the structure of an object at runtime. When it comes to reflection, the JavaScript language doesn't offer much. The Microsoft AJAX library largely makes up for this.

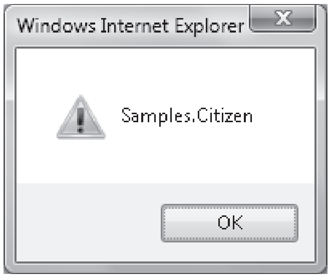
In plain JavaScript, the built-in *typeof* operator returns information about the type of the variable you are dealing with. The operator, though, is limited to the core set of JavaScript types. Let's consider the following code snippet:

```
Samples.Citizen = new function() {
   ⋮
}
var c = new Samples.Citizen();
alert(typeof c);
```

As expected, the displayed string is a generic *object*.

Adding a thick object-oriented infrastructure, the Microsoft AJAX library makes it easy to track the exact name of the pseudo-type of a given object. The following code returns a more precise message, as shown in Figure 4-2.

```
// Returns "Samples.Citizen"
var c = new Samples.Citizen();
alert(Object.getTypeName(c));
```



Samples.Citizen

FIGURE 4-2 The "real" type name of a JavaScript object

Whenever a new object is registered with the Microsoft AJAX framework, its name and pseudo-type are added to an internal list. Reflection functions just look up these internal dictionaries and return what they read.

Note I use the expression *pseudo-type* to indicate a type that has its own fully qualified name according to the Microsoft AJAX library, such as *Person* in the preceding code snippet. It should be noted, though, that at the lower level of the JavaScript engine there remains a plain *object* type.

In the Microsoft AJAX library, reflection capabilities are offered as extensions of the *Type* object. These methods enable you to collect information about an object, such as what it inherits from, whether it implements a particular interface, and whether it is an instance of a particular class. Note that the *Type* class aliases the built-in JavaScript *Function* object. Therefore, many of the methods exposed through the general interface of the *Type* object are also available through the instance of any custom type (that is, function) you create.

Table 4-3 lists the members of the *Type* object, which is also a compendium of the reflection capabilities of the Microsoft AJAX library.

TABLE 4-3 Members of the *Type* Object

Member	Description
callBaseMethod	Invokes a base class method with specified arguments
getBaseMethod	Gets the implementation of a method from the base class of the specified instance
getBaseType	Gets the base type of the specified instance
getInterfaces	Returns the list of interfaces that the type implements
getName	Gets the name of the type of the specified instance
implementsInterface	Indicates whether a given instance implements the specified interface
inheritsFrom	Indicates whether the type inherits from the specified base type
initializeBase	Invokes the base constructor of a given type
isClass	Indicates whether the specified type is a Microsoft AJAX library class
isImplementedBy	Indicates whether the specified interface is implemented by the object
isInstanceOfType	Indicates whether the object is an instance of the specified type
isInterface	Indicates whether the specified type is an interface

TABLE 4-3 **Members of the *Type* Object**

Member	Description
isNamespace	Indicates whether the specified object is a namespace
Parse	Returns an instance of the type that is specified by a type name
registerClass	Registers an object as a Microsoft AJAX library class
registerEnum	Registers an object as a Microsoft AJAX library enumeration
registerInterface	Registers an object as a Microsoft AJAX library interface
registerNamespace	Creates a namespace

Finally, here's a brief example of how to use reflection in practice:

```
var t = Samples.Components.Timer;
var obj = new Samples.Components.Timer();
if (obj.isInstanceOfType(t))
{
   alert(t.getName() + " is a " + obj.getName() + ".");
}
```

The Application Object

The execution of each Web page that links the Microsoft AJAX library is controlled by an application object. This object is an instance of a private class—the *Sys._Application* class. An instance of the application object is created in the body of the library, specifically in the *MicrosoftAjax.js* file:

```
// Excerpt from MicrosoftAjax.js
Sys.Application = new Sys._Application();
```

If properly initialized, the application object invokes a pair of page-level callbacks with fixed names—*pageLoad* and *pageUnload*:

```
function pageLoad(sender, args)
{
   // sender is the Sys.Application instance
   // args   is of type Sys.ApplicationLoadEventArgs
   ⋮
}
function pageUnload(sender, args)
{
   // sender is the Sys.Application instance
   // args   is of type Sys.ApplicationLoadEventArgs
   ⋮
}
```

In particular, *pageLoad* is a good place for the page to perform any initialization tasks that require the Microsoft AJAX library. This is more reliable than using the *window*'s *onload* event.

The *pageLoad* callback receives a *Sys.ApplicationLoadEventArgs* data structure packed with the list of Microsoft AJAX library components already created and a Boolean flag to indicate that the callback is invoked within a regular postback or a partial rendering operation.

Beyond page loading events, the *Sys.Application* object serves one main purpose: providing access to client-side page components. Generally, the term *component* denotes an object that is reusable and can interact with other objects in the context of a framework. In the Microsoft AJAX framework, a component is a JavaScript object that inherits from the *Sys.Component* class. These objects are tracked by the library infrastructure and exposed via methods on the *Sys.Application* object.

In particular, the *findComponent* method scrolls the run-time hierarchy of Microsoft AJAX components for the current page until it finds a component with a matching ID. The method has two possible prototypes:

```
Sys.Application.findComponent(id);
Sys.Application.findComponent(id, parent);
```

The former overload takes the ID of the component, uses it to look up the component, and then navigates the hierarchy all the way down from the root. When a non-null *parent* argument is specified, the search is restricted to the subtree rooted in the context object. The *id* parameter must be a string; the *parent* parameter must be a Microsoft AJAX library object. The method returns the object that matches the ID, or it returns null if no such object is found.

The Microsoft AJAX library also supports a shortcut for retrieving run-time components—the *$find* method. The *$find* method is an alias for *findComponent*:

```
var $find = Sys.Application.findComponent;
```

You can use this method to locate all components created by server controls that use the Microsoft AJAX library (for example, extenders in the AJAX Control Toolkit and new controls in ASP.NET 4.0), as well as by your own JavaScript code. You can't use *$find* to locate DOM elements; for DOM elements, you must resort to *$get*.

String Manipulation

The *Sys.StringBuilder* class adds advanced text manipulation capabilities to Web pages based on the library. As the name suggests, the class mimics the behavior of the managed *StringBuilder* class defined in the .NET Framework.

When you create an instance of the builder object, you specify initial text. The builder caches the text in an internal array by using an element for each added text or line. The *Sys.StringBuilder* object doesn't accept objects other than non-null strings. You add text using the *append* and *appendLine* methods. The *toString* method composes the text by using the join method of the JavaScript array class.

```
// Build an HTML table as a string
var header = "<table><thead> ... </thead>";
var footer = "<tfoot> ... </tfoot></table>";
var builder = new Sys.StringBuilder(header);
⋮
builder.append(footer);
alert(builder.toString());
```

The Microsoft AJAX library *String* class is also enriched with a format method that mimics the behavior of the *Format* method on the .NET Framework *String* class:

```
alert(String.format("Today is: {0}", new Date()));
```

You define placeholders in the format string using *{n}* elements. The real value for placeholders is determined by looking at the *n*.th argument in the format method call.

Debugging

Another class that is worth mentioning is the *Sys._Debug* class. An instance of this internal class is assigned to the *Sys.Debug* global object:

```
Sys.Debug = new Sys._Debug();
```

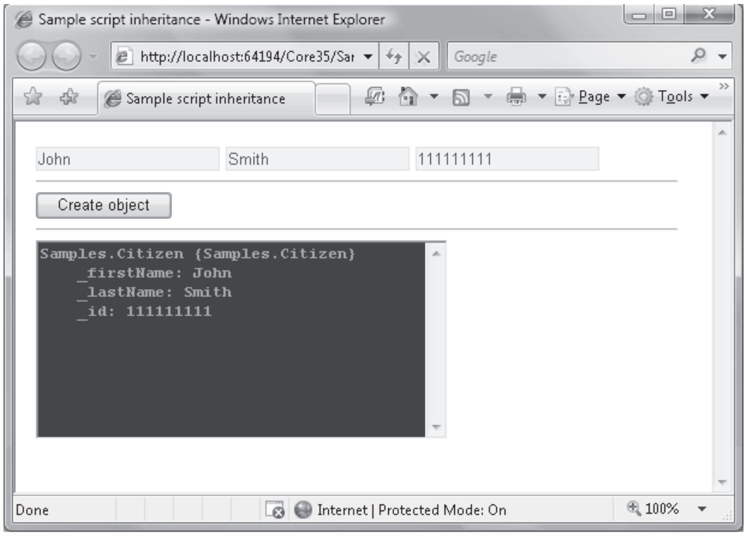
In your pages, you use the *Sys.Debug* object to assert conditions, break into the debugger, or trace text. For example, the *traceDump* method writes the contents of the specified object in a human-readable format in the Microsoft AJAX library trace area. The trace area is expected to be a <*textarea*> element with a mandatory ID of *traceConsole*. You can place this element anywhere in the page:

```
<textarea id="traceConsole" cols="40" rows="10" />
```

The *traceDump* method accepts two parameters, as shown here:

```
Sys.Debug.traceDump(object, name)
```

The *name* parameter indicates descriptive text to display as the heading of the object dump. The text can contain HTML markup. Figure 4-3 shows the output of a trace dump.


Sample script inheritance - Windows Internet Explorer
http://localhost:64194/Core35/Sar
Google
Sample script inheritance
Page
Tools
John
Smith
111111111
Create object
Done
Internet | Protected Mode: On
100%

```
Samples.Citizen {Samples.Citizen}
    _firstName: John
    _lastName: Smith
    _id: 111111111
```

FIGURE 4-3 The Microsoft AJAX library debugging tracer in action

You use the *clearTrace* method to clear the output console. The *fail* method breaks into the debugger and the method *assert* displays a message if the specified condition is false.

The Network Stack

The most relevant feature of an AJAX library is the ability to execute out-of-band Web requests from the client browser. In particular, AJAX extensions to ASP.NET let you invoke Web service methods without dismissing the currently displayed page. This ability leverages the networking support built into the Microsoft AJAX library.

In the Microsoft AJAX library, a remote request is represented by an instance of the *Sys.Net.WebRequest* class. Table 4-4 lists the properties of the class.

TABLE 4-4 Members of the *Sys.Net.WebRequest* Object

Member	Description
body	Gets and sets the body of the request
executor	Gets and sets the Microsoft AJAX library object that will take care of executing the request
headers	Gets the headers of the request
httpVerb	Gets and sets the HTTP verb for the request
timeout	Gets and sets the timeout, if any, for the request
url	Gets and sets the URL of the request

The *WebRequest* class defines the *url* property to get and set the target URL and the *headers* property to add header strings to the request. If the request is going to be a POST, you set the body of the request through the *body* property. A request executes through the method *invoke*. The *completed* event informs you about the completion of the request.

Each Web request is executed through an internal class—the Web request manager—that employs an "executor" to open the socket and send the packet. All executors derive from a common base class—the *Sys.Net.WebRequestExecutor* class.

The Microsoft AJAX library defines just one HTTP executor—the *Sys.Net.XMLHttpExecutor* class. As the name suggests, this executor uses the popular *XMLHttpRequest* object to execute the HTTP request.

The *Sys.Net.WebRequest* class is essentially a framework class that other higher level classes use, but page authors hardly ever use it. I've seen this class used only a few times in real-world JavaScript code. As you saw in Chapter 2, the ASP.NET AJAX framework makes it so easy to invoke a Web service method or perhaps a static method on a page that you hardly feel the need to invoke another type of HTTP endpoint.

If you need to download a resource such as a JavaScript file, you need quite a bit of code if you go through this class.

```
var endpoint = "ondemand.js";
var request = new Sys.Net.WebRequest();
request.set_url(endpoint);
request.add_completed(function() {...});
request.invoke();
```

With other AJAX libraries—for instance, jQuery—this code reduces to just one line. I'll return to jQuery in the next chapter.

Note All AJAX libraries are associated with the *XMLHttpRequest* browser object. So what else could an executor be other than a reference to the *XMLHttpRequest* browser object? In general, an HTTP executor is any means you can use to carry out a Web request. An alternative executor might be based on HTTP frames. The idea is to use a dynamically created inline frame to download the response of a given request and then parse that result into usable objects.

The Eventing Model

Building cross-browser compatibility for events is not an easy task. Internet Explorer has its own eventing model, and so do Firefox and Safari. For this reason, the event model of the Microsoft AJAX library is a new abstract API that joins together the standard W3C API and the Internet Explorer model. The new API is closely modeled after the standard W3C API.

In addition to using different method names (*add/removeEventListener* is for Firefox, and *attach/detachEvent* is for Internet Explorer), browsers differ in the way they pass event data down to handlers. In Internet Explorer, an event handler receives its data through the global *window.event* object; in Firefox, the event data is passed as an argument to the handler. In the Microsoft AJAX library, event handlers receive a parameter with proper event data.

Another significant difference is in the way mouse and keyboard events are represented. The Microsoft AJAX library abstracts away any differences between browsers by providing ad hoc enumerated types, such as *Sys.UI.Key* and *Sys.UI.MouseButton*. Here's some sample code:

```
function Button1_Click(e)
{
  if (e.button === Sys.UI.MouseButton.leftButton)
  {
    ⋮
  }
}
function keyboard_EnterPressed(e)
{
  if (e.keyCode === Sys.UI.Key.enter)
  {
    ⋮
  }
}
```

The Microsoft AJAX library provides a shorthand notation to create DOM event hookups and removal. For example, you can use the *$addHandler* and *$removeHandler* aliases to add and remove a handler. Here's the syntax:

```
$addHandler(element, "eventName", handler);
$removeHandler(element, "eventName", handler);
```

In many cases, you'll want to hook up several handlers to a DOM event for a component. Rather than manually creating all the required delegates and related handlers, you can use a condensed syntax to add and remove multiple handlers:

```
initialize: function()
{
   var elem = this.get_element();
   $addHandlers(
         elem,
         {[
            'mouseover': this._mouseHoverHandler,
            'mouseout': this._mouseOutHandler,
            'focus', this._focusHandler,
            'blur', this_blurHandler
         ]},
         this);
}
```

The *$clearHandlers* alias, conversely, removes all handlers set for a particular DOM element in a single shot.

If you write a component and wire up some events, it's essential that you clear all handlers when the component is unloaded, or even earlier, if you don't need the handler any longer. For example, you should do that from the component's *dispose* method to break circular references between your JavaScript objects and the DOM. Correctly applied, this trick easily prevents nasty memory leaks.

Summary

JavaScript is one of the pillars of the Web. Now that the arrival of AJAX is shaking the foundation of the Web, what about JavaScript? Is JavaScript going to change in the near future?

For years, the JavaScript language has remained very stable, and this stability created the environmental conditions for AJAX to flourish and thrive. AJAX means more and more code hosted and running within the client browser. This code can only be written in JavaScript.

The perception of a language is different when you have only a few lines of code to write as opposed to when you have to use it to write large sections of the application. For this more exacting job, JavaScript seems more inadequate every day. And JavaScript 2.0 is slowly but steadily emerging. JavaScript 2.0 is not a thing of the immediate future, though.

For now, a better and richer JavaScript is possible only through libraries that cover the parts of client-side programming that the language doesn't natively cover. Classes, networking, static type checking, and a common and cross-browser model for managing events and exploring the document are all features required in modern JavaScript code. Popular libraries, such as the Microsoft AJAX library, provide just this.

The key trait of the Microsoft AJAX library is the set of extensions to transform JavaScript into an object-oriented language. JavaScript is not a true OOP language even though it always has supported objects and also provides a rudimentary mechanism for prototyping classes and derived classes. The Microsoft AJAX library builds on top of this basic functionality to add namespace and interface support in addition to a number of helpful facilities.

In the next chapter, I'll cover another extremely popular library that addresses UI enhancements and makes it so easy and effective to add AJAX capabilities to Web pages. This library is jQuery.

Index

Symbols and Numbers

# (hash) symbol, 168, 236, 245
# (pound) symbol, 33
# symbol, 143
#id selector, 142
#PropertyName expression, 241, 245
#Quote placeholder, 236
#Style1 expression, 248
#Word expression, 247–48
$ .getScript method, 176
$ function, 59, 140–41
  helper methods, 141–42
  parameters, 140
$ root object, 115
$ shorthand notation, 159
$addHandler alias, 126
$clearHandlers alias, 126
$create shortcut, 263
$find method, 122
$G function, 94
$get function, 114–15
$removeHandler alias, 126
:animated()filter, 145
:button filter, 148
:checkbox filter, 148
:checked filter, 148
:contains(text) filter, 146
:disabled filter, 148
:empty filter, 146
:enabled filter, 148
:eq(index) filter, 144–45
:even filter, 144
:file filter, 148
:first filter, 144
:first-child filter, 145
:gt(index) filter, 144
:has(selector) filter, 146
:header()filter, 145
:hidden filter, 148
:image filter, 148
:input filter, 148
:last filter, 144
:last-child filter, 145
:lt(index) filter, 144
:not(selector) filter, 144
:nth-child filter, 145
:nth-child(expression) filter, 145
:odd filter, 144
:only-child filter, 145
:parent filter, 146
:password filter, 148
:radio filter, 148
:reset filter, 148
:selected filter, 148
:submit filter, 148
:text filter, 148
:visible filter, 148
. . . (ellipses), 291
* (asterisk) symbol, 263
* selector, 143
.asmx, 65
.asmx endpoints, 32
.asmx requests, 76, 78
.aspx source file, 55
.class selector, 142
.js files, 110
.NET Base Class Library, 286
  LINQ support, 286–87
  thread support, 287–88
  timer support, 289
.NET Framework
  classes, as data source, 230
  classes, Silverlight and, 23
  code security, 302
  data binding, 223–24
  data transfer to JavaScript, 232–33
  DataContractSerializer class, 166
  enumerations, 114
  isolated storage, 289–90, 308. *See also* storage
  jQuery library, 137, 139
  Managed Jscript compiler, 102
  presentation platform, 23
  security attributes, 306
  Silverlight, 274–75, 279, 308
  StringBuilder class, 122
  Web Forms model, 19
  XBAP applications, 276
.NET Framework 2.0, 75
.NET Framework 3.5
  binding, 68
  HTTP endpoint, 64
  LINQ-to-SQL, 166
  Service Pack 1, 33, 66, 78, 266
  WCF services, 66–67, 72
.svc endpoint, 32
.svc file, 74
.svc requests, 72, 78
/js prefix, 80
/jsdebug prefix, 80
@Page directive
  Culture attribute, 32
  UI culture attribute, 31
@ServiceHost directive, 74
_ (underscore) symbol, 118
_firstName members, 108
_initialize method, 40
_lastName members, 108
[. . .] notation, 117
[attribute != value] filter, 146
[attribute $= value] filter, 146
[attribute *= value] filter, 146
[attribute ^= value] filter, 146
[attribute = value] filter, 146
[attribute] filter, 146
~~ operator, 166
+ (plus) character, 265
<body> tag, 263
<div> dialog method, 161
<div> element, 21, 185
<div> tag, 44, 248
  as template, 261
  HTML template, 240
<domain> element, 301
<form> element, 143
<grant-to> node, 301
<h2> element, 144
<img> tag, 77–78
  Same Origin Policy, 216
<input> elements, 143–44, 148
  color, 149
  extenders, 58
  widgets, 59–60
<object> tag, 277
  plug-ins, 273
<p> element, 144
<script> tag, 29, 77–78, 139
  cross-domain requests, 301
  cross-site scripting attacks, 85
  Same Origin Policy, 216
  script downloads, 176–78
<select> element, 148
<span> element, 144
<span> tag, 47, 205
  color, 265
<table> tag, 205
<tbody> element, 262
<td> element, 146–47, 248
<textarea> element, 123, 148
= = = operator, 107
3D API, 276
3D graphics, 277
401 errors, 86

A

abortPostBack method, 208
abstract base classes, 55
access, unauthorized, 85–86
Accordion widget, 160
ActionLink component, 260
Active Server Pages (ASP), 88, 276
ActiveX, 270
  security, 272–73
Add member, 230
addClass function, 150
administrator permissions, 302–03
ADO.NET Data Services, 21, 78–79
  ASP.NET library, 266–68
  container classes, 230
  content display, 267–68
  HTTP façade, 66–67
  proxy class, 267
  Silverlight, 300
Adobe, 272
Adobe Flash, 22, 271
  cross-domain requests, 301
  plug-ins, 274
AdoNetDataSource object, 268
AJAX, 3, 221
  API, script downloads, 176
  architecture, 61–62. *See also* architecture
  ASP.NET 4.0 data binding, 260–68
  aspects, 94–96
  benefits, 11, 18
  connectivity, Yahoo! UI library, 136
  costs, 7, 18
  data binding, 232
  defined, 12
  design patterns. *See* AJAX design patterns
  history feature, 33–34, 167–68
  implementation, 27
  infrastructure, 28–37
  JavaScript injections, 53–60
  JavaScript library, 35–37
  jQuery library functions, 157–58
  library. *See* Microsoft AJAX library
  network stack, 67
  page script manager, 28–34
  paradigm change, 27
  partial rendering, 37–53, 61. *See also* partial rendering
  PrototypeJS library, 133–34
  pure applications, 49
  refactoring to, 16–17
  Silverlight, 308
  templates, 260–66
  vs. Silverlight, 23, 275
  WCF services, 73–74
  Web and, 4–11
  Web site development, 17
AJAX 4.0, 260–68
AJAX Control Toolkit, 57–59, 258
  obtaining, 57
AJAX design patterns, 163, 213, 221
  applying, 164
  code development, 163–68
  Cross-Domain Proxy pattern, 215–17
  dynamic data download, 166
  generalities, 164–66
  Heartbeat pattern, 193–94
  idioms, 165–66
  JavaScript development, 168–78
  Micro-Link pattern, 213–15
  On-Demand JavaScript pattern, 175–78
  page DOM manipulation, 166–67
  Periodic Refresh pattern, 192–93
  Predictive Fetch pattern, 178–86
  Progress Indicator pattern, 194–213
  Singleton pattern, 169–70
  Submission Throttling pattern, 218–20
  Timeout pattern, 186–92
  unique URL, 167–68
  user actions, 167
  value of, 164–65
AJAX Extensions 1.0, 75
Ajax.request function, 133–34
Ajax.Updater function, 134
AjaxOptions settings, 260
AjaxSetting element, 93
aliases, 115
  findComponent, 122
  Function object, 21
  handler addition and removal, 126
  handlers, 126
  jQuery object, 159
Always value, 44
  exceptions, 45
ancestor descendant selector, 143
animate function, 156–57
animate() call, 157
animation, Silverlight, 278
anonymous functions, 116
asynchronous requests, Silverlight, 295–96
API (application programming interface). *See* application programming interface (API)
append method, 122
appendLine method, 122
appHeader class, 144
application code, 306–07
  safe critical code, 304–06
application object, 121–22
application programming interface (API), 27
  3D, 276
  browser, 168
  caching, jQuery, 158
  client progress, 196, 200–05, 211–12
  custom JavaScript functions, 113–14
  DOM, 233
  dual, 24
  event handling, 151–55
  HTTP façade, 62–63
  jQuery library, 137
  server, 197–98
  Silverlight communication, 296–98
  task server, 198
  W3C, 125
applyFormatting method, 248
architecture, 61–62, 97–98
  data binding, 224–34
  HTTP façade, 70–79
  multitier, 21
  patterns, 61–62, 163. *See also* Server Pages pattern; Service Layer pattern
  presentation layer, 79–82
  Silverlight, 275–77
  Web, 18–19
Aristotle, 163
Array object, 112
arrays, 117
  custom, 230
  interfaces, 119
  wrapped sets, 141
as keyword, 166
ASCX, template creation, 229
ASMX Web services, 21
ASP (Active Server Pages), 88, 276
ASP.NET, 12
  compatibility mode, 87
  controls, Server Pages pattern, 88–89
  controls, YUIAsp.Net library, 135
  data binding, 224–34. *See also* data binding
  data sources, 230
  flexibility, 21–22
  Forms authentication, 86–87
  future of, 18–19, 69
  IPostBackDataHandler interface, 41
  JavaScript, 130–31
  JavaScript libraries, 104

jQuery library, 137–39
layers, 22
library, 266–68
panels, 39
partial rendering. *See* partial rendering
ScriptManager. *See* ScriptManager control
server controls, 69
server-side Web development, 130–31
template properties, 46
templates, 260–66
timeouts, 187–88
UpdateProgress control, 195
view state, 42
WCF services, 72
Web service, 65–67
Web services, HTTP façade, 75–78
ASP.NET 2.0
control state, 48
partial rendering, 28
view state, 42
ASP.NET 3.5
CompositeScript collection, 175
Microsoft AJAX library, 110–11
partial rendering, 28
UpdateProgress control, 195
ASP.NET 4.0, 69, 79, 88
data binding, 260–68
Microsoft AJAX library, 110
ASP.NET AJAX. *See* AJAX
ASP.NET AJAX Extensions 1.0, 110
ASP.NET MVC
applications, 70
HTML Message pattern demo, 259–60
jQuery library, 137–38
aspect ratio, 160
aspects, 94–96
Aspects property, 95–96
AspNetCompatibilityRequirements attribute, 87
Assembly object, 298
AssemblyPart class, 298
assert method, 124
asterisk (*) symbol, 263
ASX formats, 278
async parameter, 157
async postback, 38
Asynchronous JavaScript And XML. *See* AJAX
asynchronous pages, 46–47
asynchronous postbacks, 45
authentication and authorization, 46
canceling, 52
Gaia, 94
multiple, 52–53
partial rendering, 52–53
UpdateMode property, 44
user feedback, 50–52
asynchronous requests, 87–88, 91–92
Partial Fetch method, 181–82
Silverlight, 289
AsyncPostBackSourceElementID property, 30
AsyncPostBackTrigger, 45
attr function, 150
attribute filters, 143, 146–47
audio formats, 278
authentication, 10
ASP.NET Forms, 85–87
asynchronous postbacks, 46
cookie, 85–86
HTTP and, 83
authorization, 10
asynchronous postbacks, 46
HTTP and, 83
AutoComplete extender, 218–19
autocompletion, 218–19
automated factories, HTML, 225
autonomous views, 171
autopostback feature, 39
autosave, 192
AvailableFreeSpace property, 294

B

Back/Forward navigation, 34
BackBase, 24, 97
BackgroundWorker class, 288
bandwidth
BST vs. HM pattern, 253
conserving, 187, 223
Periodic Refresh pattern, 192
streaming, 278
Base CSS file, 135
basicHttpBinding, 75, 84
BeginGetResponse method, 298
BeginInvoke method, 288
BeginRead/BeginWrite method, 292
BeginRequestStream method, 298
behaviorConfiguration attribute, 74
Betamax, 136
Bibeault, Bear, 150
binary code
ActiveX, 270
Flash, 271
interoperability, 270–71
over Web, 270–71
security, 270–71
BinaryReader class, 293
BinaryWriter class, 293
bind function, 152
bind method, 241, 244–45
binding. *See* data binding
Binding class, 284
Binding element, 285
black box, 88–89, 131
blur(fn) helper, 154
body member, 124
body property, 124
Boolean object, 112
Boolean parameters, 70
Boolean properties, 44, 208
MVC pattern, 175
Boolean values, 295
brackets
curly, 262
JSON and XML, 69
browser, 62
API, 168
basic model, 8–9
cross-browser compatibility, 18, 125–26, 132, 150
high-end, 15
history, 167–68
HTML data conversion, 233
JavaScript and, 102–05
navigation, 33–34, 167–68
optimization, 6–7
plug-ins. *See* plug-ins
Same Origin Policy (SOP), 66, 215–16
sandbox, 274
scripting engine, 103–04
session termination, 187–88
technologies, 12
wars, 1990s, 8
Browser Object Model (BOM), 13
JavaScript dependencies, 110
Browser-Side Template (BST) pattern, 133–34, 235, 268
data and template mixing, 236–37
dual-side templating, 237–38
HTML builder, 241–45
HTML template, 236
item rendering, customized, 245–47
jQuery library effects, 249–50
markup rendering, customized, 247–48
page preparation, 239–41
reference implementation, 238–50
vs. HTML Message pattern, 252–53
BST (Browser-Side Template) pattern. *See* Browser-Side Template (BST) pattern
buffer, 245
Builder design pattern, 140

builder script, 135
building applications, Silverlight, 279–81
bulleted lists, 228
business layer, 22, 63–64
business logic, 97
  HTML Message pattern, 252
  protecting, 83, 85
Button control, 94, 283–84
  disabling, 48
button handlers, 175
bytecode, 104

C

C#, 19
  ~~ operator, 166
  code-behind class, 280
  HM pattern, 253
  keywords, 286
  namespace{. . .} construct, 116
  objects in, 36, 105–06
  Silverlight, 22
  Singleton pattern, 169
C++, 19
  objects in, 105–06
cache parameter, 157
caching, 158
  client-side, 178
  Predictive Fetch pattern, 183–86
  size, performance, 198
Calendarextender control, 57–59
call chaining. *See* chaining
callback functions, 80–82, 149, 157
  arguments for, 81
  Partial Fetch method, 181–82
  script downloads, 176
callback manager, 92
callback parameter, 215
callbacks
  fade effects, 249
  GetCurrentStatus method, 204–05
  HTML template, 245
  JavaScript, 246–47
  pageLoad and pageUnload, 121
  startMonitor method, 202, 205–06
  success and failure, 204–05
  timer, 289
  updateProgress function, 205
callBaseMethod, 117
callBaseMethod member, 120
calls. *See* requests
camelCase, 156
Cancel button, 51–52, 207–08, 211
cancellation requests, 195
CanRead property, 292
Canvas element, 282–83
CanWrite property, 292
CAS (Code Access Security) model, 302–03
Cascading Style Sheets (CSS), 10
  attributes, 155–56
  future of, 37
  hiding HTML, 236
  property naming convention, 156
  selectors, 142–44
  server controls, 131
  Server Pages pattern, 88
  user feedback, 50
  Yahoo! UI library, 135
CERN, 8
CGI, 21
chaining, 151
  animations, 157
  call, 137
  jQuery library, 137, 139
Change method, 289
Change property, 247, 265
change(fn) helper, 154
CheckBox control, 283
CheckBox control (Gaia), 94
CheckBoxList control (Gaia), 94
child controls, 44–45
  adding, programmatic, 46
  page methods, 55
child filters, 145
ChildrenAsTriggers property, 44
circular references, 126
Citizen class, 116–18
classes, 36. *See also* specific classes
  ADO.NET container, 230
  as data source, 230
  bindable, 230
  code-behind. *See* code-behind class
  creating, Dojo syntax, 134
  CSS, 150
  custom, 230
  data readers, 230
  derived, 117, 127
  finder, 255–57
  formatter, 105
  in C# and C++, 105–06
  inheritance rules, 307
  interfaces, 118–19
  JavaScript, 36–37
  JSON, 68
  managed, 67
  object-oriented extensions, 115–16
  proxy, 79–82
  renderer, 255–57
  Service Layer pattern, 64
  Silverlight, 79
  Singleton pattern, 169–70
  template creation, 229
  timer, 190
  Web services, proxy, 67
clearTrace method, 124
click event, 39, 152, 202, 207
  registering, 154
  triggering, 153
click handler, 211
click(fn) helper, 154
ClickOnce technology, 276
client browser. *See* browser
client certificates, 10
client side, 20
  API, 200–05, 211–12
  BST pattern, 133–34
  caching, 158, 178, 183–86
  Cancel button, 51–52
  cancellation request, 195
  controls, 89–90
  data binding. *See* data binding
  data transfer, 232–33
  focus, 61
  JavaScript, 28
  JavaScript API, 196
  monitoring, Timeout pattern, 187
  object model, 90, 92, 95–96
  out-of-band requests, 124–25
  page-component access, 122
  power, JavaScript, 101
  programming, JavaScript library, 35
  task ID, 198–99
  task interruption, 211–12
  task termination, 211–12
  user actions, 167
  Web development, 130
ClientAccessPolicy.xml file, 301
ClientBin Web server folder, 281
Close method, 292
closure model, 107–08
CLR (Common Language Runtime). *See* Common Language Runtime (CLR)
code
  access security, 302
  application, 304–06
  application vs. platform, 306–07
  binary. *See* binary code
  BST vs. HM pattern, 252–53
  bytecode, 104
  compatibility, 279
  compatibility, WPF, 285–86
  critical, 303–04
  development, design patterns, 163–68
  flexibility, 233
  GZIP compression, 136
  idioms, 165–66

interpreted, 104
jQuery minimization, 137
machine, 110
malicious, 105
managed, 90
manual iteration, 149
Microsoft AJAX library, 119
minification, 136
MVC pattern improvements, 170
performance and, 109
safe vs. unsafe, 303
safe-critical, 304–06
sample, plug-ins, 159
security, Silverlight. *See* code security, Silverlight
server side, 66–68, 252
Silverlight, 279–80
source, 104–05
source code file pairs, 282
transparency, 302–03
Code Access Security (CAS) model, 302–03
Code activity, 212–13
code group, 302
code security, Silverlight, 302
access security, 302
application code vs. platform code, 306–07
code transparency, 302–03
critical code, 303–04
design security, 306–07
inheritance rules, 307
safe-critical code, 304–06
security attributes, 303–06
security model, 302–03
code-behind class, 21–22
HTTP façade, 64
page methods, 54–55
partial rendering, 41
Server Pages pattern, 88, 90
Silverlight, 279–80
source code file pairs, 282
collections, custom, 230
color values, 248
ColorAnimation class, 278
combo boxes, 92–93
Common Language Runtime (CLR)
Silverlight, 24
Silverlight instancing, 281
Silverlight security, 302. *See also* CoreCLR
compatibility
.NET and Silverlight, 279
code, 279, 285–86
code, WPF, 285–86
cross-browser, 18, 125–26, 132, 150
JavaScript changes, 101, 108–09
JSON vs. XML, 69
compatibility mode, 78, 87
compiled programming languages, 104–05
compilers
AJAX solutions, 24
JavaScript, 110
Managed Jscript, 102
completed event, 124
Component Object Model (COM), 273
ComponentArt, 18, 96–97
components, 122
CompositeScript collection, 175
concatenation, 146
concurrent requests, 92
partial rendering, 52–53
conditional updates, 44–45
Conditional value, 44
config file, 87
constructors, 117
initialize method, 134
interfaces, 118
jQuery object, 140
templates, 242
container classes, 230
container element, 114–15
content filters, 146
Content property, 284
ContentTemplate property, 46
ContentTemplateContainer property, 46
content-type headers, 77
context argument, 81
context parameter, 186
context, query, 140–41
contextKey parameter, 259
contract attribute, 74
contracts, 80
progress server API, 209
control state, 48
controller classes, 175
controller, MVC pattern, 171–72
controls. *See also* server controls
button, 89
design time, 88–89
script, 135
Controls collection, 46
cookies, 85–86, 289
Dojo library, 133
Yahoo! UI library, 136
CopyTo member, 230
CoreCLR, 274, 302–03
application code vs. platform code, 306–07
code transparency, 303
safe-critical code, 304–05
Count member, 230
Create, Read, Update, Delete (CRUD) operations, 267, 291
CreateFile method, 292
createInstance method, 262
critical code, 303–04
Crockford, Douglas, 109, 136
cross-browser compatibility, 18, 125–26, 132, 150
Cross-Domain Proxy pattern, 215
error handling, 217
Same Origin Policy, 215–16
Web remoting via JavaScript, 216
cross-domain requests, Silverlight, 300–01
CrossDomain.xml file, 301
Cross-Domain-Proxy pattern, 257
cross-page method calls, 55
cross-site scripting attacks, 77–78, 85, 216
CRUD operations, 267, 291
CRUDy user interface, 66
CSS (Cascading Style Sheets). *See* Cascading Style Sheets (CSS)
css function, 150
Culture attribute, 32
Culture property, 32
culture settings, 32
CultureInfo class, 286
Current property, 56
custom arrays, 230
custom collections, 230
custom dictionaries, 230
custom events, 135
custom handlers, 78
custom HTML tags, 236
custom item rendering, 245–47
custom markup rendering, 247–48
custom objects, 230
custom services, 78

D

data
binding. *See* data binding
caching, 158
contract, 75
deletion, 294
downloads. *See* downloads
encapsulation, JavaScript, 107–08
exchange, HTTP façade, 66
exchange, Service Layer pattern, 62–63
fetching. *See* Predictive Fetch pattern
formats, Service Layer pattern, 67–68
items, 43
preloading, 178

data *(continued)*
  serialization, 20–21
  server-to-client transport, 232–33
  services, 78–79
  source controls, 21–22
  source, definition, 230–34
  storage. *See* storage
  storage, Dojo library, 133
  transfer latency, 178
  transfer minimization, 47–48
  transfer, Service Layer pattern, 67–68
data access layer, 22
data attribute, 277
  <object> tag, 274
data binding, 68, 73–74, 223–24, 268
  AJAX, 232
  ASP.NET, 224–34
  ASP.NET 4.0, 260–68
  Browser-Side Template (BST) pattern, 235–50. *See also* Browser-Side Template (BST) pattern
  data source definition, 230–34
  HTML Message pattern, 250–60. *See also* HTML message pattern
  HTML template, 224–30
  jQuery library, 152
  partial rendering, 234
  properties, 231–32
  Silverlight programming, 284–85
  tools for, 232–33
data function, 157–58, 184
data parameter, 215
data property, 152
data reader classes, 230
data source component, 267–68
data source properties, 223
DataBind method, 231
data-bound controls, 227
  properties, 231–32
data-bound items, 239–41
DataContext property, 285
DataContract attribute, 75
DataContractSerializer class, 166
Data-for-Data model, 15–16
DataGrid control, 225, 237, 246, 282–83
DataKeyField property, 232
DataList control, 226
DataMember attribute, 75
DataMember property, 232
DataServiceContext class, 79
DataSet class, 230–31
DataSets, typed, 21–22
DataSource property, 231–32
DataSourceID property, 231–32
DataTable class, 230–31
DataTextField property, 232
dataType parameter, 157
DataValueField property, 232
DataView class, 230
DataView component, 262–64, 268
Date data types, 32
Date object, 112
date picker widget, 57–59, 160
DatePicker control, 282–83
dblclick(fn) helper, 154
de Saint-Exupery, Antoine, 101
debug mode, 80
debug script files, 31
debugging, 123–24
  reflection, 119
Deep Zoom, 278–79
Deep Zoom Composer, 279
defaultSucceededCallback property, 81
defaultUserContext property, 81
deferred loading, 267
delta, 89–90
  size, 48
derived classes, 117, 127
deserialization, 75
design patterns. *See* AJAX design patterns
Design Patterns (Gamma, Helm, Johnson, and Vlissides), 164
design-time controls, 88–89
desktop applications
  AJAX and, 3–4
  Rich Internet Applications and, 22–24
developers, 6–7
DHTML (Dynamic HTML), 13, 281
dialog box creation, 161
dialog method, 161
Dialog widget, 160
Dickens, Charles, 269
dictionaries
  as objects, 105–06
  custom, 230
directories, storage, 291–92
DirectoryInfo class, 291–92
disabling of visual elements during updates, 52
discrete interpolation, 278
disk quotas, 294–95
Dispatcher object, 288
display attribute, 214
DisplayAfter property, 51–52
displayCustomer function, 185
DisplayName property, 285
dispose method, 126
Document Object Model (DOM), 8–10, 12–13
  $get method, 122
  API, 233
  circular references, 126
  CLR instancing, 281
  handler hookup and removal, 126
  ID, 143
  JavaScript dependencies, 110
  JavaScript downloads, 176
  JavaScript library, 35–36
  On-Demand JavaScript pattern, 177–78
  page manipulation, 166–67
  queries, 140–41
  retrieval, 143
  script downloads, 177–78
  smooth page updates, 42–43
  visual effects. *See* visual effects
document.getElementById function, 95, 114–15, 135
documentation files, 140
Documents and Settings folder, 291
Dojo library, 36, 133
  JS Dojo Minifier, 136
domain attribute, 301
DoSomething function, 287
dot notation, 106
DoubleAnimation class, 278
downloads, 166, 175–76
  binary content, 297
  Silverlight, 296–98
  XAP packages, 281
DownloadStringAsync method, 296
DownloadStringCompleted event, 296
DoWork event, 288
drag-drop script, 135
draggable interaction, 160
DropDownList control, 39
droppable interaction, 160
dual application programming interface, 24
Dual-Side Templating pattern, 254
duck typing, 107
dynamic data download, 166
Dynamic HTML (DHTML), 13, 281
dynamic programming languages, 106
dynamic table, 205–06
dynamic templates, 46
dynamic typing, 106
DynamicPopulate extender, 258–59

E

each method, 149
each(callback) method, 142
ECMA (European Computer Manufacturers Association), 103
ECMAScript, 103
EF (Entity Framework), 79, 230, 300
effects script, 135

Eich, Brendan, 103
elem1 variable, 145
elem2 variable, 145
element selector, 142
elements
  adding/removing, 151
  aspects, 160
  caching, 158
  chaining, 151
  hidden, 148
  interactivity, 159–61
  loop processing, 149
  selecting, 143–44
  Silverlight layout manager, 282–83
  visual effects, 155–57
  wrapping, 149–50
Elements of User Experience, The (Garrett), 5
ellipses (. . .), 291
EnableHistory property, 34
EnablePageManager attribute, 200
EnablePageMethods property, 55
EnablePartialRendering property, 30
EnableScriptGlobalization property, 32
EnableScriptLocalization property, 31
enableWebScript attribute, 80
enableWebScript element, 73
encapsulation, 37
  JavaScript, 107–08
endpoints, 21
  .svc file, 74
  event sink, 199–200
  HTTP. *See* HTTP endpoints
  IIS, 86
  page, 70
  periodic refresh, 192
  public service, 33
  script downloads, 176
  service, 70
endsWith method, 113
entity data model, 79
Entity Framework (EF), 79, 230, 300
entry points, static, 170
enumerations, 180–81
eq(position) method, 142
error handling, 217
Error object, 113
error(fn) helper, 154
errors
  code 401, 86
  handling, 217
  interfaces, 119
Esposito, Dino, 172
European Computer Manufacturers Association (ECMA), 103
European Organization for Nuclear Research, 8
eval function, 21, 67, 69, 176
event handlers, 125. *See also* handlers; HTTP handlers; REST handlers
  adding/removing, 115
  grouping, controller class, 175
  JavaScript, 61–62
  MVC pattern, 172
  postbacks, 47
  server side, 93
  threads, 288
event handling, 151–55
event object properties, 153
event scheduling, 196
event sink, 196, 199–200
eventing model, 125–26
events
  as idioms, 166
  custom, 135
  helpers, 154
  keyboard, 125, 135, 188, 191
  mouse, 125, 135, 188
exceptions
  CoreCLR, 304
  data binding, 231
  directories and files, 291–92
  disk quota, 295
  duck typing, 107
  invalid operation, 231
  not-implemented, 118
  parameter count, 117
  RequirementsMode, 87
  Update method, 45
ExecuteTask method, 202
executor class, 208
executor member, 124
executors, 124
  HTTP, 125
Exists property, SecuritySafeCritical attribute, 305
explicit contracts, 75
Explicit Submission, 220
explicit triggers, 44–45
extenders, 57–59
extensibility, jQuery library, 139
extensions. *See* JavaScript language extensions; object-oriented extensions
ExtJS library, 133

F

factories
  HTML templates, 229–30
  HTML, automated, 225
  HTML, template-based, 225–27
Factory attribute, 74
fadeIn function, 155
fadeOut function, 155
fadeout method, 250
fadeTo function, 155
fading effects, 156
fail method, 124
failure callbacks, 204–05
file streams, storage, 292–93
FileInfo class, 291–92
FileMode value, 292
files, storage, 291–92
filters, 144–48
  attribute, 143, 146–47
  child, 145
  content, 146
  positional, 144–45
FinanceInfoservice, 81
findComponent method, 122
findCustomer function, 181
finder class, 255–57
FindQuoteInfo method, 257
Firefox, 125
  Dojo library compatibility, 133
  NAPI, 272
  prototypes, 108
  textElement property, 150
firewall, 85
FishEye control (Gaia), 94
Flash, 271
  cross-domain requests, 301
  plug-ins, 274
flexibility, 233
Flush method, 292
focus(fn) helper, 154
footer
  HTML template, 228, 239–41
  list items, 226, 237
foreach construct, 166
form filters, 148–49
form submission, 167, 220
Format method, 123
Format property, 57–58
formatter classes, 67
forms, 14–15
Franklin, Benjamin, 27
free content, 217
Function class, 117
Function object, 112–13, 115
  aliases, 120
function(i), 149
functions. *See also* specific functions
  AJAX, jQuery library, 157–58
  anonymous, 116
  as objects, 106
  event handling, 151–55
  JavaScript, 106
  JavaScript, custom, 113–14

functions *(continued)*
named, 116
registration, 114
shorthand, 114–15, 135
timer, 190
visibility, 155–56
vs. methods, jQuery library, 159

G

Gaia AJAX library, 97
Gaiaware, 18, 94–96
Gamma, Erich, 164
Gandhi, Mahatma, 61
Garrett, Jesse James, 3–5, 7
GenerateHtml method, 257
GenerateItem method, 227
Get call, 220
get function, 158
GET request, 215
GET verb, 72–73, 77–78, 158
security, 84
get() method, 142
get(index) method, 142, 145
getBaseMethod member, 120
getBaseType member, 120
GetCurrentStatus method, 200, 204
GetCustomerDetails method, 260
GetData member, 231
GetDirectoryNames method, 291
getElementById method, 143
getElementsByName method, 143
GetFileNames method, 291
getInstance method, 170
getInterfaces member, 120
getJSON function, 158
getName member, 120
GetQuotes method, 81
GetQuotesFromConfig method, 70–71, 263
GetStatus method, 198
getTaskID method, 202
GetUserStoreForApplication method, 290, 293–94
GetUserStoreForSite method, 290
global members, 108
Global object, 135, 169
globalization, 32, 286
Google, 24
Google Chrome browser, 104–05, 110
Google V8 engine, 105, 110
Google Web Toolkit, 24, 97
graphics, Silverlight, 277–79
grid element, 241
caching, 158
Grid element, 282–83
grids, 160
GridView control, 246
GridView control, disabling, 48
GUID, 199, 202
GZIP compression, 136
jQuery library, 138

H

handlers. *See also* event handlers; HTTP handlers; REST handlers
adding and removing, 126
binding/unbinding, 152
button, 175
custom, 78
factory, 76–77
IPostBackDataHandler interface, 41
MVC pattern, 174–75
onclick, 174–75
OnTick, 189
partial rendering, 40–41
REST, 76
handles, 160
hash (#) symbol, 168, 236, 245
hash strings, 33–34
header
HTML template, 228, 239–41
list items, 226
headers, 43
concurrent call issues, 52
content-type, 77
HTTP request, 70
headers member, 124
headers property, 124
Heartbeat pattern, 193–94
Helm, Richard, 164
helper methods, $ function, 141–42
helpers
event, 154
parameters, 154
hidden elements, 148
hidden HTML, 236
template, 261
hide function, 155
history feature, 33–34, 167–68
history point addition, 33–34
hosting applications, plug-ins, 274
hover function, 153
HTML, 10–11
binding/unbinding, 152
browser/server basic model, 8–9
builder, BST pattern, 241–45
changing/replacing, 10–11
custom tags, 236
Data-for-Data model, 15
Document Object Model (DOM). *See* Document Object Model (DOM)
factories, 225–27, 229–30
generating, 257–58
hiding, 236
JavaScript limitations, 108–09
limitations, 49
origins, 8
Pages-for-Forms model, 14–15
plain type, 248
plain, future of, 37
presentation layer. *See* presentation layer
purpose of, 9
renderer, 257–58
Rich Internet Applications, 23
script files, linking to page, 30–31
standardization, 10
string creation, 233, 235
table, 205–06
tables, 225–26
template, 224–30, 236
template-based factories, 225–27
templates, hidden, 261
HTML 4.0 DOM standard, 40, 114
HTML 4.0 standard, 270
html function, 150
HTML Message pattern, 134, 250–51, 268
ASP.NET MVC demo, 259–60
Dual-Side Templating pattern, combined, 254
DynamicPopulate extender, 258–59
markup rendering, 257–58
motivation for, 252
reference implementation, 253–60
remote service, 254–57
vs. BST pattern, 252–53
HtmlListBuilder class, 239–41, 250
HTTP, 10–11
401 error code, 86
binding, 73–74
Data-for-Data model, 15–16
direct calls, 54–56
executor, 124–25
origins, 8
purpose of, 9
requests. *See* requests
security, 10, 83
statelessness, 14
verbs, 66, 78. *See also* specific verbs
HTTP endpoints, 30, 64, 66–67, 70, 124
HTML Message pattern, 252
requests, 70–71
script downloads, 176
HTTP façade, 62–64, 70–79
ADO.NET services, 78–79

ASP.NET Web services, 75–78
custom services, 78
HTTP request, 70–71
proxy, 80–82
security, 83–84
Server Pages pattern, 90
Service Layer pattern, 64
technologies for, 65–67
trusting, 86–87
WCF services, 72–75
HTTP handler
custom, 66–67, 78
factory, 76–77
load method, 215–16
HttpContext, 56
HttpContext.Current object, 78
HTTPS, 10, 83
httpVerb member, 124
HyperlinkButton control, 283
hyperlinks, 213–14

I

ICollection interface, 230
id parameter, 122
ID property, 117
IDE (integrated development environment), 131
idioms, 165–66
IDisposable interface, 118
IEnumerable interface, 230–31
IFinanceInfoRenderer interface, 257
IFRAME elements, 133
IIS (Internet Information Services). *See* Internet Information Services (IIS)
IList interface, 230
IListSource interface, 230–31
images
Deep Zoom, 278–79
JPG and PNG, 279
impedance mismatch, 67
implementsInterface member, 120
implicit contracts, 75
implicit triggers, 44–45
IncreaseQuotaTo property, 294–95
independent software vendors (ISVs), 69–70
index(element) method, 142
information architecture, 5
Infragistics, 18
inheritance, 36–37, 116–18, 307
inheritsFrom member, 120
Init event, partial rendering, 41
initialize method, 134
initializeBase member, 120
initializeBase method, 117
inner text, 150
innerHTML property, 42, 150, 233, 235, 254
DynamicPopulate extender, 258
fade effects, 250
HTML Message pattern, 251
scripting, 248
innerText property, 150
input controls, 57–59
input elements, 61–62. *See also* <input> elements
disabling, 52, 137, 150
InstantiateIn method, 227
integer functions, 149
integrated development environment (IDE), 131
IntelliSense, 116
jQuery library, 139
interactivity, 167
JavaScript, 132
jQuery UI, 159–61
widgets, 132
interfaces, 118–19
Internet Explorer, 103, 125
closures and prototypes, 108
Dojo library compatibility, 133
GZIP, 138
innerText property, 150
Internet Explorer 4.0, 13
Internet Explorer 5.0, 13
Internet Explorer 8, XDomainRequest object, 301
Internet Explorer 8.0, 168, 216
Internet Information Services (IIS)
endpoints, 86
HM pattern, 253
Smooth Streaming, 278
WCF services, 72
interoperability. *See also* compatibility
ActiveX, 271
finder and renderer classes, 255
JavaScript, 108–09
JSON vs. XML, 68
Web binary code, 270–71
interpolation, 278
interpreted code, 104
interpreted programming languages, 104
interruptible server tasks, 208–10
invoke function, 259
invoke method, 124
IronPython, 280
IronRuby, 280
ISAPI, 21
isClass member, 120
isImplementedBy member, 120
isInstanceOfType member, 120
isInterface member, 120
isNamespace member, 121
isOffline argument, 263
isOffline parameter, 70
isolated storage
safe critical code, 306
Silverlight, 289–95
IsolatedStorageFile class, 290–92
IncreaseQuotaTo property, 294–95
Quota property, 294
IsolatedStorageFileStream class, 292–93
ISVs (independent software vendors), 69–70
itemCreated event, 264
ItemDataBound event, 237
ITemplate interface, 50, 227–29
items, customized rendering, 245–47
ItemSource property, 285
Iterator pattern, 166
ITransaction interface, 213

J

Java Server Pages, 276
Java, vs. JavaScript, 11
JavaScript, 10, 12, 101–02, 126–27, 129–30
AJAX design patterns, 168–78. *See also* specific patterns
AJAX injections, 53–60
ASP.NET, 130–31
callbacks, 246–47
changing/replacing, 10–11
client API, 196
compatibility, changes and, 104–05
compiler, 110
data binding. *See* data binding
data transfer from .NET, 232–33
Data-for-Data model, 16
flaws, 104–05
future of, 37, 108–10, 126–27
Google V8 engine, 110
GZIP compression, 136
hiding from view, 28
HTML builder, 236–39, 241–45
injections, polling, 54
interactivity, 132
language, 109
language and browser, 102–08
libraries. *See* JavaScript libraries; Microsoft JavaScript client library
mashups, 216
minification, 136
Model-Controller pattern, 82
MVC pattern, 172–75

JavaScript *(continued)*
  MVC pattern sample, 172–75
  object-orientation addition, 36–37
  On-Demand pattern, 175–78
  overhaul, 109
  parsing, 235
  presentation layer. *See* presentation layer
  pros and cons of, 36
  proxy, 33, 73, 75, 80–81
  Rich Internet Applications, 23–24
  security, 109
  selective updates, 18
  server controls, 131–32
  Server Pages pattern, emissions increasing, 89–90
  Silverlight, 280
  standardization, 10
  syntax, 49
  timers, 219–20
  types, 106–07
  variables, 107
  versions, 36
  vs. Java, 11
  Web remoting, 216
  widgets, 56–60
  Yahoo! UI library, 135–36
JavaScript 2.0, 109, 126
JavaScript language extensions
  new, 113–14
  primitive, 112–13
  shorthand functions, 114–15
JavaScript libraries, 105, 127, 129–30, 161
  Dojo library, 133
  ExtJS library, 133
  jQuery library, 137–61. *See also* jQuery library
  list of, 132–33
  memory requirements, 133
  popular, 132–33
  PrototypeJS library, 133–34
  Script.aculo.us library, 134–35
  selection, 136–37
  server controls and widgets, 130–37
  size, 136
  synthesis of features, 136–37
JavaScript Notation. *See* JSON
Johnson, Ralph, 164
JPG images, 279
jQuery function, 140–41
jQuery in Action (Bibeault, Katz, and Resig), 150
jQuery library, 36, 137–38, 161
  $ function, 140–42
  $ root object, 115
  AJAX functions, 157–58
  benefits of, 136–37
  BST pattern, 249–50
  caching, 158, 184
  core library, 140–42
  downloading, 138
  fundamentals, 138–39
  jQuery UI, 138
  methods vs. functions, 159
  micro-links, 215
  obtaining, 59
  plug-ins, 139, 158–59
  Predictive Fetch pattern, 180
  selectors, 142–49
  size, 138
  UI framework, 159–61
  utilities, 151–61
  visual effects, 155–57
  Visual Studio 2008, 139–40
  widgets, 138
  wrapped sets, 141, 149–51
jQuery object, 140–41
  alias, 159
  CSS selectors, 142–43
  helper methods, $ function, 141–42
  plug-ins, 158–59
  wrapped sets, 141
jQuery UI
  downloading, 161
  widgets, 160
jQuery.fn object, 159
JS Dojo Minifier, 136
Jscript, 103
JSMin, 136
JSON, 232
  endpoints, 21
  page methods, 54
  serialization, 62, 67–68, 77
  services, invoking, 33
  strings, 21, 67–68, 70–71
  vs. XML, 20–21, 67–69
  Web services, 76–77
JVC, 136

K

Katz, Yehuda, 150
keyboard events, 125, 135
  timeouts, 188, 191
keydown(fn) helper, 154
keypress event, 219
keypress(fn) helper, 154
keyup(fn) helper, 154
keywords, 286
Konqueror, 272

L

Label control, 89, 94
Label1 control, 285
LastName, FirstName string, 117
last-win discipline, 53
latency, 193
layers, ASP.NET, 22
layout manager, 282–83
lblWait element, 260
Left property, 283
legitimate users, 83
length method, 142
Length property, 141, 292
libraries, 97, 168–69
  .NET Base Class Library, 286–89
  ASP.NET, 266–68
  binary, 273. *See also* plug-ins
  Gaiaware's Gaia AJAX, 94–96
  JavaScript. *See* JavaScript libraries
  jQuery. *See* jQuery library
  Microsoft AJAX. *See* Microsoft AJAX library
  RadControls library, 92–94
  Server Pages pattern, 92–96
  YUIAsp.Net library, 135
linear interpolation, 278
links, 213–14
  micro-links. *See* micro-links
LINQ, .NET Base Class Library, 286–87
LINQ-to-JSON, 286–87
LINQ-to-Objects, 286
LINQ-to-SQL, 166, 230, 267
LINQ-to-XML, 286–87
Linux platform, 271, 277
list items, 226, 228
ListBox control, 283
ListView control, 226
Live Form pattern, 220
Livequotes.svc, 263
LiveScript, 103
Load event, 41
load function, 158
load method, 216
  micro-links, 215
load times, 108
load(fn) helper, 154
Loaded attribute, 280
loadFooter method, 244
loadHeader method, 244
loading, deferred, 267
loadItemTemplate method, 244
local storage, 290
localeFormat method, 32
logical tokens, 290
login page, 85–86
London, Jack, 129

looping
  plug-ins, 158–59
  Repeater control, 225–27
  wrapped sets, 149
Lowy, Juval, 75

M

Mac platform, 271
machine code, 110
managed classes, 67
managed code, 90
Managed Jscript compiler, 102
managers, 7
manual iteration, 149
mapping, MVC pattern, 172
markup, 80, 87–90, 92
  ASP.NET, 130
  asynchronous postbacks, 94
  BST vs. HM pattern, 252–53
  customized rendering, 247–48
  DataGrid control, 225
  HTML Message pattern, 251
  HTML templates, 226–28
  incomplete/invalid, 236
  load function, 158
  object dump, 123
  RadControls, 94
  rendering, HTML Message pattern, 257–58
  server controls, 131
  server side, 252
  Silverlight, 279–80
  source code file, 282
  startMonitor task, 205–06
  template creation, 229–30
Markup slot, 158
mashups, 215, 223
  error handling, 217
  JavaScript, 216
master page, ScriptManager, 29–30
match method, 245
Math object, 202
media formats, 278
media pack, Silverlight, 277–78
MediaElement control, 277–78
members. *See also* specific members
  global and static, 108
  overridable, 117
  private, 118
  public vs. private, 108
  Sys.Net.WebRequest class, 124
  Type object, 120–21
  var keyword, 108
Memento pattern, 166
memory
  cache size, 186
  JavaScript libraries, 133
  Predictive Fetch pattern, 179
metadata, service, 73
methodCompleted callback, 55–56
methodName argument, 81
methods. *See also* specific methods
  ad hoc, 291
  MVC pattern, 172
  overriding, 117, 307
  static, 170, 290
  vs. functions, jQuery library, 159
Micro-Link pattern, 213–15
micro-links, 213–14
  actions, 214
  jQuery library, 215
Microsoft, 270
Microsoft .NET: Architecting Applications for the Enterprise (Esposito and Saltarello), 90, 172
Microsoft AJAX library, 89, 110, 127
  constituent files, 110–11
  framework facilities, 119–26
  JavaScript language extensions, 112–15
  linking, 111–12
  object-oriented extensions, 115–19
  Singleton pattern, 169–70
Microsoft JavaScript client library, 28
  downloading, 35
  files of, 35–36
  object-orientation addition, 36–37
  pros and cons, 36
Microsoft Metadirectory Services (MMS), 278
Microsoft Product Support Services, 137–38
Microsoft Visual Studio 2008, IntelliSense, 116
MicrosoftAjax.js, 35
MicrosoftAjax.js file, 111, 121
MicrosoftAjaxTemplates.js, 261
MicrosoftAjaxTimer.js, 35, 111
MicrosoftAjaxWebForms.js, 35, 111
middle tier
  HTTP façade, 63–64
  security, 87
  unauthorized access, 85
minification, 136, 159
MMS (Microsoft Metadirectory Services), 278
modality, 95
Mode12, 171
model, MVC pattern, 171–72
Model-View-Controller (MVC) pattern, 82, 170–75
  HM pattern demo, 259–60
Model-View-Presenter (MVP) pattern, 90, 171
monitoring, progress indication, 195–96
Mono project, 277
monolithic views, 171
Moonlight group, 277
mouse events, 125, 135
  timeouts, 188
mousedown(fn) helper, 154
mousemove(fn) helper, 154
mouseout(fn) helper, 154
mouseover(fn) helper, 154
mouseup(fn) helper, 154
MP3 format, 278
multimedia, Silverlight, 277–79
Multistage Downlaod pattern, 175
multithreading, 219–20
MVC (Model-View-Controller) pattern. *See* Model-View Controller (MVC) pattern
MVP (Model-View-Presenter) pattern, 90, 171
MyControllerClass, 260
MyPet function, 106
MySourceObject class, 285

N

name parameter, 123
Name property, 75, 80
  Microsoft AJAX library, 111
named functions, 116
namespace, DataView class, 263
Namespace property, 75, 80
namespace{. . .} construct, 116
namespaces, 36–37
  JavaScript, 109
  object-oriented extensions, 115–16
naming conventions. *See also* shorthand notation
  brackets, 69
  camelCase, 156
  files, 291
  plug-in file, 159
  semicolon, 159

NAPI (Netscape Plug-in Application Programming Interface). *See* Netscape Plug-in Application Programming Interface (NAPI)
Navigate event, 34
navigation, 33–34, 167–68
nesting, 46
Netscape, 103, 272
Netscape Navigator, 113, 272
Netscape Navigator 2.0, 10
Netscape Plug-in Application Programming Interface (NAPI), 272
  security, 272–73
network stack, 67, 124–25
networking, Silverlight, 295–301
new operator, 106
not-implemented exceptions, 118
npruntime, 272
null values, 166
Number data types, 32
Number object, 112–13
NumericUpDown extender, 58

O

Object object, 112–13
ObjectAnimationUsingKeyFrames object, 278
ObjectDataSource control, 231
object-oriented extensions
  classes, 115–16
  inheritance and polymorphism, 116–18
  interfaces, 118–19
  namespaces, 115–16
object-oriented languages, 36, 102, 104–06, 127
  ASP.NET, 130–41
  JavaScript, 36–37
objects. *See also* specific objects
  as dictionaries, 105–06
  bindable, 230
  circular references, 126
  custom, 230
  functions as, 106
  global, 169
  JavaScript, 36, 105–06
  JSON strings, 67–68
  plain type, 119–20
  pseudo-type, 119–20
  reflection, 119–21
ObservableCollection type, 285
Observer pattern, 166
OLE2, 273
onclick handlers, 174–75
onDataAvailable parameter, 81
On-Demand JavaScript pattern, 175–78
one function, 152
one method, 153
OnSearchComplete callback function, 181
OnSearchComplete function, 185
OnTick handler, 189
OpenFile method, 292
OpenReadAsync method, 297
Opera, 272
OperationContract attribute, 74
operators, JavaScript, 106
optimization, 6–7
options argument, 160
origin, 216
outsiders, 83
  protecting from, 85–86
overriding, methods, 117, 307

P

P/Invoke subsystem, 303
packages, JavaScript, 109
page. *See* Web page
Page class
  Culture property, 32
  UICulture property, 31
page endpoints, 70
page methods, 54–55
  as page endpoints, 70
  event sink, 199–200
  invoking, 55–56
  vs. services, 56
Page.ClientScriptRegisterArray Declaration method, 202
pageLoad, 121
PageMethods class, 55–56
PageMethods proxy, 202
PageRequestManager class, 170, 208
PageRequestManager object, 43, 208
Pages-for-Forms model, 14–15
pageUnload, 121
pageX property, 153
pageY property, 153
panel refresh
  conditional updates, 44–45
  programmatic updates, 45
paradigm shift, 14
parameters
  $ function, 140
  ajax function, 157
  application object, 122
  constructor, 117
  helpers, 154
  integer, 149
  interfaces, 119
  load method, 215
  registerClass method, 116
  traceDump method, 123
  unbind method, 153
  visibility functions, 155
parent > child selector, 143
parent argument, 122
parent class, 55
parent parameter, 122
Parse member, 121
parser, XML, 68–69
parsing, JavaScript, 235
partial page updates, 129
partial rendering, 27
  concurrent call issues, 52–53
  data binding, 234
  disabling of visual elements during updates, 52
  limitations, 48–53, 61
  mechanics, 40–41
  motivation, 38
  page updates, 42–43
  polling, 53
  postback model, 91–92
  Predictive Fetch pattern, 181
  Server Pages pattern, 88
  server-side, 41–42
  size of, 48
  syntax, 38–39
  task cancellation, 207
  UpdateProgress control, 195
  user feedback, 49–52
  view state, 42
  vs. Server Pages pattern, 91–92
partial trust Web client, 63–64
partial view state, 95
Path attribute, 31
Path class, 291
Path property, 81–82, 111
patterns. *See also* AJAX design patterns; specific patterns
  architectural, 61–62
  architectural vs. design, 163
performance
  cache size, 198
  Data-for-Data model and, 16
  JavaScript, 108–09
  services vs. page methods, 56
Periodic Refresh pattern, 192–93
permissions, 302–03
Person class, 117
PHP Hypertext Preprocessor, 276
piecemeal submission, 218–19
placeholders, 123, 236, 238
  Dual-Side Templating pattern, 238
  HM pattern, 254
plain-old XML (POX), 72
plain-type HTML, 248
plain-type objects, 119–20
platform code, 306–07

plug-ins, 271
  <object> tag, 273
  characteristics, 273
  downloading, 158
  file naming conventions, 159
  history, 272
  hosting applications, 274
  jQuery library, 137, 139, 158–59
  jQuery UI, 159–61
  Rich Internet Applications, 22
  security, 272–73
  Silverlight, 23
plus (+) character, 265
PNG images, 279
pnlDetails element, 260
Point property, 278
PointAnimation class, 278
polling, 51
  partial rendering, 53
  Timeout pattern, 186–87
polymorphism, 116–18
PopupButtonID property, 57–58
ports, HTTP/HTTPS, 83
Position property, 292
positional filters, 144–45
post function, 158
POST request, 215, 220
POST verb, 73, 77, 158
  WebRequest class, 298
PostAuthenticateRequest event, 72
postback model, 90–91
  partial rendering, 91–92
postbacks, 19
  asynchronous. *See* asynchronous postbacks
  autopostback feature, 39
  conditional updates, 44–45
  delta, 90
  headers, 43
  lengthy tasks, 46–47
  model. *See* postback model
  partial rendering, 40–41
  queue, 52
  server controls, 89
  Server Pages pattern, 87–88
  triggers, 91–92
  view state, 42
pound (#) symbol, 33
POX (plain-old XML), 72
Predictive Fetch pattern, 178–79
  caching, 183–86
  reference implementation creation, 180–86
  sample scenario for, 180–81
PreInit event, 46
PreRender event handler, 41
presentation layer, 62–63, 79, 97
  commercial frameworks, 57
  HTTP façade proxy, 80–82
  JavaScript Model-Controller pattern, 82
  MVC pattern, 170–71
presentation logic, 250
  separating from user interface, 235
preventDefault method, 153
previous ~ sibling selector, 143
previous + next selector, 143
private members, 118
productivity
  Predictive Fetch pattern, 179
  server controls, 88, 131–32
programmatic updates, 45
programming languages, 102–05. *See also* object-oriented languages; specific languages
  idioms, 165–66
  Silverlight, 280
Programming WCF (Lowy), 75
progress bar display, 196, 205
Progress Indicator pattern, 194–213
  ASP.NET UpdateProgress control, 195
  reference implementation, 196–213
  sample scenario, 196–213
progress meter, 205
progress template, 207–08
ProgressChanged callback, 288
ProgressMonitor class, 197–98
ProgressTemplate property, 50
PropertyName property, 241
Prototype, 36
Prototype library, 105
prototype model, 107–08
  Microsoft AJAX library, 116
prototype object, 106, 108
prototype property, 37, 106
  Dojo library, 134
PrototypeJS library, 133–34
prototypes
  findComponent method, 122
  interfaces, 118–19
  overridable, 117
ProviderName property, 257
proxy
  ADO.NET Data Services, 267
  classes, 79–82
  classes, Web services, 67
  JavaScript, 73, 75, 80–81
  JavaScript, Web remoting, 216
  page methods, 55–56
  remote services invoking, 33
  REST handler, 80
  ScriptManager, 29–30
pseudo-type object, 119–20
Python, 24

Q

queries, 140
query chaining. *See* chaining
query method, 267
Query Object pattern, 166
queue:false flag, 157
QuickTime, 272
Quota property, 294
Quote property, 247

R

RadAjaxManager control, 93–94
RadControls library, 97
RadioButton control, 283
RadioButtonList control (Gaia), 94
RandComboBox, 93–94
reach, 22
  vs. rich, 23, 308
Read/Write method, 292
read-only properties, 117
ready function, 153
RealPlayer, 272
Real-Time Streaming Protocol (RTSP), 278
refactoring, 16–17
reference implementation, 180–86
  Browser-Side Template (BST) pattern, 238–50
  HTML Message pattern, 253–60
  Progress Indicator pattern, 196–213
  Timeout pattern, 188–92
reflection, 119–21
  example, 121
refresh. *See* page updates
RegExp object, 113
RegisterASyncPostBackControl method, 45
registerClass member, 121
registerClass method, 116–17, 119
registerEnum function, 114
registerEnum member, 121
registerInterface member, 121
registerNamespace member, 121
registerNamespace method, 116
registration functions, 114
release script files, 31
remote methods, 54–56
remote requests, 124–25, 181–83
remote services
  HTML message pattern, 254–57
  invoking, 33
remote tasks, canceling, 206–08
remote URLs, 251
Remove member, 230
removeClass function, 150
removeData function, 158, 186

renderer class, 255–57
RenderMode property, 47
Repeater control, 225–27
replaceWith method, 150
replay attacks, 85
ReportProgress method, 288
Representational State Transfer (REST), 66
request-for-markup Web model, 252
requests
  ajax function, 157
  asynchronous, 87–88, 91–92, 166
  cancellation, 195
  concurrent, 92
  cross-domain, Silverlight, 300–01
  HTTP endpoints, 70–71
  out-of-band, 124–25
  partial rendering, 52–53
  payloads, 94–95
  remote, 124–25, 181–83
  Silverlight, 298–99
  synchronous, 296
RequestTermination method, 209–10
RequirementsMode property, 87
Reset CSS file, 135
Resig, John, 59, 138, 150
resize(fn) helper, 154
resizeable interaction, 160
ResourceUICultures property, 31
response time, 178–79
responsiveness, 178
REST (Representational State Transfer), 66
REST handler, 76, 84
  proxy class, 80
REST services, 21, 267
RESTful interface, 78
results argument, 81
RIA (Rich Internet Application). *See* Rich Internet Application (RIA)
rich, vs. reach, 308
Rich Internet Application (RIA), 17, 269, 308
  binary code, 270–71
  browser plug-ins, 271–74
  HTML and, 23
  JavaScript, 23–24
  Silverlight, 274–81. *See also* Silverlight
rich user experience, 56–57
richness, vs. reach, 23
RichTextBox control, 283, 286
rollbacks, 212–13
RowDataBound event, 246
RTSP (Real-Time Streaming Protocol), 278
Ruby, 24
Ruby on Rails, 133
run-time
  errors, interfaces, 119
  HTML templates, 227
  JavaScript, 103–04
  modules, headers, 43
Russell, Bertrand, 223

S

S function, 135
Safari, 125
  Dojo library compatibility, 133
  NAPI, 272
safe-critical code, 304–06
  application code, 304–06
Saltarello, Andrea, 172
Same Origin Policy (SOP), 66, 105
  Cross-Domain Proxy pattern, 215–16
  Silverlight, 300–01
Samples.HtmlListBuilder object, 241–45
Samples.Person function, 116
Samples.Progress class, 202–04
Samples.Services, 81
Samples.Services.FinanceInfoService, 81
sandbox, 105
save point, 168
scalability, postbacks, 47
script code/files
  cross-site scripting attacks, 77–78, 85
  debug and release, 31
  download, 175
  globalization, 32
  JavaScript library, 35
  linking, 30–31
  loading files, 157
  localizable elements, 31
  optional and custom, loading, 30–31
  sandbox, 105
  Script.aculo.us library, 135
  user-interface elements, 31
Script.aculo.us, 36, 133
Script.aculo.us library, 134–35
scriptaculous.js file, 134
scripting attacks, 105, 216
scripting engine, 103–04
scripting languages, 107
ScriptManager control, 28
  code for, basic, 29
  ID, 41
  jQuery library, 139
  Microsoft AJAX library, 111–12
  partial rendering, 30. *See also* partial rendering
  RadControls library, 93
  Services section, 32–33
ScriptManagerProxy, 29–30
ScriptMethod attribute, 77
ScriptModule, 55
Scripts collection, 30–31
ScriptService attribute, 75–76
scroll(fn) helper, 154
scrollbars, 283
ScrollViewer element, 282–83
Secure Sockets Layer (SSL), 83
security, 83
  ActiveX, 270–73
  binary code over Web, 270–71
  code, Silverlight, 302–07. *See also* code security, Silverlight
  countermeasures, 84–85
  GET verb, 77–78
  HTTP and, 10
  HTTP façade, 63–64
  HTTP façade, trusting, 86–87
  inheritance rules, 307
  JavaScript, 109
  middle tier, HTTP façade as, 83–84
  NAPI, 272–73
  outsiders discrimination, 85–86
  plug-ins, 272–73
  replay attacks, 85
  Same Origin Policy (SOP), 66
  timeouts, 192
security model, Silverlight, 302–03
SecurityCritical attribute, 303, 306
SecuritySafeCritical attribute, 303, 306
SecurityTransparent attribute, 303, 306
SecurityTreatAsSafe attribute, 306
Seek method, 292
select(fn) helper, 154
selectable interaction, 160
SelectIndexChanged event, 39
selective updates, 17–18
selector1, ..., selectorN selector, 143
selectors, 142
  CSS, 142–44
  filters, 144–48. *See* filters
  form filters, 148–49
  query, 140–41
separation of concerns (SoC), 170–71
serialization, 20–21, 62, 67–68
  contracts, 75
  JSON, 77
  partial rendering, 41
  view state, 42
  XML, 77
server
  basic model, 8–9
  resources, 187, 192

server code, 62
server controls, 37, 69–70, 82, 129
  ASP.NET, 88–89, 131
  data binding, 223–24
  data-bound controls, 224
  future of, 132
  Gaia AJAX library, 94–96
  JavaScript and, 131–32
  RadControls, 92–94
  Server Pages pattern. *See* Server Pages pattern
  Silverlight, 276–77
  strengths and weaknesses, 131–32
  view state, 131
Server Pages pattern, 87–88
  advantages and disadvantages, 96–97
  ASP.NET classic controls, 88–89
  code-behind and Service Layer pattern, 90
  HTTP façade, 90
  JavaScript emissions, increasing, 89–90
  libraries, 92
  postback model, classic, 90–92
  vs. partial rendering, 91–92
  vs. Service Layer pattern, 96–97
server side, 20
  API, 197–98
  ASP.NET development, 130–31
  ASyncPostBackTrigger, 45
  code, 66–68, 88–89, 252
  controls, 61–62
  data binding, 224, 232, 246
  data transfer, 232–33
  event handler, 93
  focus, 61
  HTML Message pattern. *See* HTML Message pattern
  HTTP endpoints, 64
  interruptible tasks, 208–10
  markup, 252
  mashup, 223–24
  optimization, 6
  partial rendering, 41–42
  processing overhead, 178
  security, 63–64
  session termination, 192
  storage, 289
  task ID, 199
  task termination, 208–10
service endpoints, 70
Service Layer pattern, 62–63
  HTML presentation layer, 69–70
  HTTP façade, 63–67, 70–79
  JSON vs. XML, 68–69
  presentation layer, 62–63, 79–82
  security, 83–87
  Server Pages pattern and code-behind, 90
  vs. Server Pages pattern, 96–97
ServiceContract attribute, 75, 80
service-level agreements, 217
ServiceMethod attribute, 259
services. *See* Web services
  references, adding to pages, 32–33
Services collection, 32
Services section, 32–33
setInterval function, 189–90
SetLength method, 292
sets, wrapped. *See* wrapped sets
SetStatus method, 198
setTimeout function, 190, 204
SFChange function, 95
shorthand functions, 114–15, 135
shorthand notation
  $ symbol, 159
  DOM hookup and removal, 126
ShouldTerminate method, 209–10
show function, 155
Silverlight, 3, 308
  .NET Framework, 279
  animation, 278
  applications, running, 276–77
  architecture, 275–77
  building applications, 279–81
  CLR, 24
  code and markup, 279–80
  Deep Zoom, 278–79
  graphics and multimedia, 277–79
  HTML and, 23
  instancing, CLR, 281
  media pack, 277–78
  platform compatibility, 271
  plug-ins, 274
  programming. *See* Silverlight programming
  programming languages, 280
  Rich Internet Application (RIA), 274–81
  security, 84
  Service Layer pattern, 97
  storage. *See* storage
  ToolTips, 282
  vs. AJAX, 275
  vs. smart clients, 276
  WCF, 75
  XAP packages, 280–81
Silverlight 1.0, 271
Silverlight 2,
    Visual Studio, 79
Silverlight programming, 282
  .NET Base Class Library, 286–89
  code security, 302–07
  common controls, 283–84
  cross-domain requests, 300–01
  data binding manager, 284–85
  data downloads, 296–98
  isolated storage, 289–95. *See also* storage
  layout manager, 282–83
  networking, 295–301
  services, 299–300
  storage. *See* storage
  Web requests, 298–99
  WPF code compatibility, 285–86
  WPF-based user interface, 282–86
Singleton pattern, 169–70
slideDown function, 155
Slider control, 283
Slider extender, 58
slider script, 135
Slider widget, 160
slideToggle function, 155
slideUp function, 155
sliding effects, 156
smart client, 64, 276
Smooth Streaming, 278
SOAP, 65, 68, 72
  ASP.NET services, 76–77
  basicHttpBinding model, 75
  disabling clients, 77
  vs. REST, 84
SoC (separation of concerns), 170–71
Sony, 136
SOP (Same Origin Policy). *See* Same Origin Policy (SOP)
sortable interaction, 160
source code
  JavaScript, 105
  scripting engine, 104
source parameter, 277
Source property, 284
SqlDataSource control, 231
src attribute, 30
SSL (Secure Sockets Layer), 83
StackPanel element, 282–83
standardization, 8, 17, 271
  HTML, 10
  JavaScript, 10
  Web structure and, 10–11
  XMLHttpRequest object, 301
startMonitor method, 202
startsWith method, 113
startTask method, 202
state objects, 14
statements, 227
static entry points, 170
static members, 108
static methods, 170, 290
static properties, 81
stopMonitor method, 202
stopPropagation method, 153
stopTask callback, 190
storage
  disk quotas, 294–95

storage *(continued)*
  file streams, 292–93
  files and directories, 291–92
  local, 290
  management, 293–94
  Silverlight, 289–95
  system, Silverlight, 290–91
stored procedures, 196
Stream object, 297, 299
streaming, 278
StreamReader class, 292–93
StreamWriter class, 292–93
strict mode, 109
String class, 123
String data type, 32
String object, 112–13
StringBuilder object, 233
strings
  comma-separated, 31
  event name, 152
  hash, 33–34
  HTML, creation, 233, 235
  JSON. *See* JSON
  manipulation, 122–23
  plug-ins, jQuery library, 158–59
  serialization, 20–21
Submission Throttling pattern, 218
  Explicit Submission pattern, 218–19
  Live Form pattern, 220
  timers, multithreading, 219–20
submit event, 91, 208
  partial rendering, 40–41
submit(fn) helper, 154
success callbacks, 204–05
success parameter, 157
Symbol property, 247, 262
synchronization, 176
synchronous requests, 296
sys:activate attribute, 263
sys:attach attribute, 264
Sys._Debug class, 123–24
Sys.Application object, 121–22
Sys.ApplicationLoadEventArgs, 121
Sys.Component class, 122
Sys.CultureInfo object, 32
Sys.Data.AdoNetDataSource class, 267
Sys.Data.AdoNetServiceProxy class, 267
Sys.Data.DataSource component, 265–66
Sys.Debug object, 123
Sys.IDisposable interface, 118
Sys.Net.WebRequest class, 124–25
Sys.Net.WebRequestExecutor class, 124, 208
Sys.Net.XMLHttpExecutor class, 124, 208
Sys.StringBuilder class, 122–23
Sys.StringBuilder object, 245
Sys.UI.DataView component, 262–64
Sys.UI.Key, 125
Sys.UI.MouseButton, 125
Sys.WebForms.PageRequest Manager object, 40
System.IO.DirectoryInfo class, 304–05
System.Json assembly, 287
System.Json namespace, 287
System.Net namespace, 85
system.web.extensions, 110
System.Web.Extensions assembly, 38
System.Web.Services.WebService, 78
System.Web.UI namespace, 38
sys-template style, 261–62

T

TabControl control (Gaia), 94
table element, 283
tables, 225–26
  sys-template style, 261–62
Tabs widget, 160
target property, 153
TargetControlID property, 259
task ID, 198–99
task server API, 198
tasks
  client, interrupting, 211–12
  implementation, 198–99
  monitoring, 195–96
  remote, canceling, 206–08
  rollbacks, 212–13
  server, interruptible, 208–10
TCP (Transmission Control Protocol), 8
TCP/IP (Transmission Control Protocol/Internet Protocol), 269
Telerik, 18
templates
  ASP.NET, 260–66
  constructor, 242
  creation, 229–30
  data and, mixing, 236–37
  defined, 223
  dual-side templating, 237–38
  dynamic, 46
  HTML, 224–30, 236
  HTML factories, 225–27
  ITemplate interface, 50
  logic injection, 264–65
  structure, 261–62
  Sys.Data.DataSource component, 265–66
  Sys.UI.DataView component, 262–64
  URI, 234
  URL, 74
  XML data islands, 240
Temporary Internet Files folder, 281
TerminateTask method, 212
text box, 92
  widgets, 57–59
text function, 150
Text property, 285
TextBlock element, 285
TextBox control, 283
  AJAX Control Toolkit, 57–59
  disabling, 48
TextBoxWatermark extender, 58
textElement property, 150
third-party software vendors, 69–70
this object, 149
this.each iterations, 159
Thread class, 287
ThreadPool class, 287
threads, 287–88
Time To Last Byte (TTLB), 178
timeout member, 124
Timeout pattern, 186–92
  related patterns, 192–206
  sample scenario, 188–91
timeout property, 81
timeouts, 193
  keyboard, 188, 191
  mouse, 188
timer class, 190
Timer control, 189
timers, 204
  .NET Base Class Library support, 289
  autocompletion, 219
  JavaScript, 219–20
  multithreading, 219–20
  Progress Indicator pattern, 194–95
TLS (Transport Layer Security), 83
toggle function, 153, 155
toggleClass function, 150
tokens, 290
Toolbar control (Gaia), 94
ToolTips, 282
Top property, 283
toString method, 117–18, 122
trace console element, 123
traceDump method, 123
traffic, 253
transactions, canceling, 52
TransactionScope activity, 212–13
transfer latency, 178
transformer, JavaScript, 109
Transmission Control Protocol (TCP), 8
Transmission Control Protocol/Internet Protocol (TCP/IP), 269
transparency, code, 302–03
Transport Layer Security (TLS), 83
trigger function, 152
trigger method, 153

triggerHandler function, 152–53
triggers
  child controls, 44–45
  explicit, 44–45
  implicit, 44–45
  partial rendering, 39
  postback, 91–92
  Silverlight, 285
Triggers collection, 285
TTLB (Time To Last Byte), 178
type attribute, <object> tag, 273–74
Type class, 120
Type object
  extensions, 120
  members, 120–21
type parameter, 157
type property, 153
Type.registerNamespace method, 116
typeof function, 107
typeof operator, 119
types, 106–07

U

UI layer, 22
UICulture attribute, 31
UICulture property, 31
unauthorized access, 85–86
unbind function, 152
unbind method, 153
underscore (_) symbol, 118
Unique URL pattern, 167–68
unload(fn) helper, 154
Update method
  exceptions, 45
  UpdatePanel control refresh, 45
UpdateMode property, 44
  exceptions, 45
update-panel, 91–92
UpdatePanel control, 28
  conditional updates, 44–45
  data binding, 234
  data transfer minimization, 47–48
  dynamic templates, 46
  multiple updatable panels, 42
  nesting, 46
  programmatic updates, 45
  RadControls, 93–94
  rules for, 47–48
  single, limitations of, 39
  smooth page updates, 42–43
  triggers, 47
  Update method, 45
  UpdateProgress control, 195
UpdateProgress control, 195
  progress bar display, 196
updating/updates, 166–67
  conditional, 44–45
  dynamic templates, 46
  partial rendering, 42–43
  programmatic updates, 45
  selective, 17–18
uri attribute, 301
URI templates, 234
UriTemplate property, 72–73
url member, 124
url property, 124
URLs
  ad hoc façade, 62
  ADO.NET Data Services proxy, 267
  creation, 37–38
  DataView, 265
  hash string modification, 33–34
  history, 167–68
  HTML tags, 177
  HTTP façade, 65–67
  HTTP verbs, 66
  posting data to, 298–99
  public, 86
  remote, 251
  syntax, ADO.NET, 79
  template, 74
  unique, AJAX design pattern,
    167–68
user actions, 167
user context object, 81
user credentials, 85, 87
user experience, 4–6
  for developers, 6–7
  for managers, 7
user feedback, 49–52
user inputs. *See* <input> elements;
    input elements
user interface
  CRUDy, 66
  data binding, 224. *See also* data
    binding
  HTML presentation layer, 69–70
  jQuery library, 159–61
  jQuery UI, 159–61
  MVC model, 172
  Periodic Refresh pattern, 192
  presentation logic separation, 235
  refreshing, 178–79
  responsiveness, 178
  Silverlight, WPF-based,
    282–86
  updating, 205–06
  widgets, 59–60
UserControl element, 282
UserControl root tag, 280
Users directory, 291

V

value property, extenders, 58
var keyword, 107–08
variables
  JavaScript, 107
  placeholders as, 236
verifier, JavaScript, 109
VHS, 136
View object, 260
view state, 27
  disabling, 48
  partial, 95
  partial rendering, 92
  server controls, 131
  size, 48
views
  autonomous vs. monolithic, 171
  MVC pattern, 171–72
visibility attribute, 214
visibility functions, 155–56
visibility, HTML template, 261
Visual Basic .NET
  code-behind class, 280
  keywords, 286
  Silverlight, 24
visual effects, 155–57
  jQuery library, 249–50
  Script.aculo.us library, 134–35
Visual GUI, 97
Visual InterDev 6.0, 88–89
Visual Studio, 21–22
  ADO.NET data services, 79
Visual Studio 2008
  jQuery library, 137–40
  proxy class, 80
  Service Pack 1, 139–40
  Silverlight Web services, 299
Vlissides, John, 164
Volta, 24
vsdoc file, 140

W

W3C (World Wide Web
    Consortium), 270, 301
WatermarkTextBox control, 283
WCF (Windows Communication
    Foundation). *See* Windows
    Communication
    Foundation (WCF)
Web 2.0, 3, 101
Web applications.
    See Web development
Web browser. *See* browser
Web development, 17.
    See also Web page;
    Web services
  AJAX capability, 17
  AJAX Control Toolkit, 57–59
  animations, 156–57
  architecture concerns, 18–22
  ASP.NET, 130–31
  client-side, 130
  HTML presentation layer, 69–70
  interactivity, jQuery UI, 159–61
  programming, 12

Web development *(continued)*
  Rich Internet Application. *See* Rich Internet Application (RIA)
  selective updates, 17–18
  service reference addition, 32–33
  user experience, 4–7
  visual effects, 155–57
  Web applications vs. AJAX applications, 167
  widgets, 56–59
Web Development Helper tool, 206
Web Forms API, 19
Web Forms model, 42
Web page
  DOM manipulation, 166–67
  free content, 217
  links, 213–14
  mashup, 215
  Pages-for-Forms model, 14–15
  polling, 53
  selective updates, 17–18
  services reference addition, 32–33
  slow, 175
  updatability, 13–14
  updates. *See* updating/updates
Web remoting, 216
Web requests. *See* requests
Web services
  custom, 66–67
  event sink, 199–200
  HTTP façade, 65–67
  metadata, 73
  proxy class, 80–82
  Same Origin Policy (SOP), 66
  security, 86–87
  Silverlight, 299–300
  Silverlight consumption, 299–300
  vs. page methods, 56
Web We Want (WWW), 5–7
web.config file
  ASP.NET Web services, 76
  debug mode, 80
  SOAP client disabling, 77
WebClient class, 296–98
WebGet attribute, 72–73
webHttpBinding model, 73, 84
WebInvoke attribute, 73
WebMethod attribute, 77
WebRequest class, 124, 298–99
WebUI, 96–97
widgets, 56–57, 132–35
  AJAX Control Toolkit, 57–59
  jQuery library, 59–60, 138
  jQuery UI, 160–61
  slider, 59–60
  YUIAsp.Net library, 135
Window control (Gaia), 94–95
window object, 140
window.event object, 125
Windows Communication Foundation (WCF), 21, 72–75, 232
  ASP.NET compatibility mode, 78
  event sink, 199–200
  HTML template, 241
  HTTP façade, 64, 66–68, 72–75
  page methods, 54
  proxy class, 80–82
  public services, 56
  runtime, 87
  security, 84, 86–87
  Silverlight, 75
  Silverlight consumption, 299–300
Windows Media Audio (WMA) formats, 278
Windows Media Video (WMV) formats, 278
Windows platform, 271
Windows Presentation Foundation (WPF)
  code compatibility, 285–86
  Silverlight, 23, 282
  Silverlight user interface, 282–86
  WPF Toolkit, 282
Windows Vista platform, 291
WMA (Windows Media Audio) formats, 278
WMV (Windows Media Video) formats, 278
World Wide Web (WWW). *See also* Web development; Web services
  binary code over, 270–71
  mechanics, 8–9
  origins, 8
  paradigm shift, 14–15
  paradox of, 9
  pillars of, 10–11
  restructuring, 10–11
  Rich Internet Application. *See* Rich Internet Application (RIA)
  security and, 10
  standardization, 8, 271
World Wide Web Consortium (W3C), 8, 13, 114, 125, 216, 270, 301
WPF (Windows Presentation Foundation). *See* Windows Presentation Foundation (WPF)
wrap method, 150
wrapped sets, 141, 149
  chaining, 151
  looping, 149
  methods, 159
  predefined operations, 150
wrapper classes, 266
Write method, 293
WSDL, 77

X

x:Class attribute, 280
x:Name attribute, 280
XAML, 279
XAML Browser Application (XBAP), 276
XAP packages, 277, 280–81
  download example, 297
XBAP (XAML Browser Application), 276
XDomainRequest object, 216, 301
XML
  data islands, 236, 240
  parser, 68–69
  plain-old (POX), 72
  vs. JSON, 20–21, 67–69
XMLHttpRequest object, 12–13, 125
  browser history, 168
  cross-domain calls, 216, 301
  custom handlers, 78
  Data-for-Data model, 15–16
  Dojo library, 133
  event sink, 196
  Heartbeat pattern, 193
  HTML Message pattern, 252
  jQuery library, 157–58
  micro-links, 214
  On-Demand JavaScript pattern, 176–77
  partial rendering, 15
  Periodic Refresh pattern, 192–93
  polling, 53
  Predictive Fetch pattern, 181
  PrototypeJS library, 133–34
  script downloads, 176–78
  standardization, 301
  Sys.Net.XMLHttpExecutor class, 124
  task cancellation, 208
xmlns attributes, 263
xxxEntities class, 300

Y

Yahoo! UI Compressor, 136
Yahoo! UI library, 36, 135–36
YUIAsp.Net library, 135

Z

ZIP archive, 280