

Dart for Hipsters

Fast, Flexible, Structured
Code for the Modern Web



Chris Strom

Edited by Michael Swaine

Early Praise for *Dart for Hipsters*

A fun and easy read for anyone wanting to understand what Dart is and how to use it with current generation browsers. The commentary on features planned for future releases of Dart is reason enough to buy this book.

► **Matt Margolis**

At first I was somewhat skeptical of Dart. This book made me understand its promise, gave me a good idea of its current state, and will serve as a solid reference for me to lean on.

► **Juho Vepsäläinen**

This is the first book on this exciting and promising programming language, a clear and approachable text that engages the reader and that certainly will contribute to Dart's success. I particularly liked his treatment of the functional aspects of the language and the discussion of isolates.

► **Dr. Ivo Balbaert**

Dart for Hipsters

Chris Strom

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://pragprog.com>.

The team that produced this book includes:

Michael Swaine (editor)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2012 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-937785-03-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June, 2012

Contents

| | |
|------------------------------|----|
| Introduction | ix |
|------------------------------|----|

Part I — Getting Started

| | | |
|-----|---|----|
| 1. | Project: Your First Dart Application | 3 |
| 1.1 | The Back End | 3 |
| 1.2 | HTML for Dart | 4 |
| 1.3 | Ajax in Dart | 5 |
| 1.4 | This App Won't Run | 9 |
| 1.5 | What's Next | 10 |
| 2. | Basics Types | 11 |
| 2.1 | Numbers | 11 |
| 2.2 | Strings | 11 |
| 2.3 | Booleans | 12 |
| 2.4 | HashMaps (aka Hashes, Associative Arrays) | 13 |
| 2.5 | Lists (aka Arrays) | 15 |
| 2.6 | Dates | 17 |
| 2.7 | Types | 17 |
| 2.8 | What's Next | 18 |
| 3. | Functional Programming in Dart | 19 |
| 3.1 | Anonymous Functions | 20 |
| 3.2 | First-Order Functions | 22 |
| 3.3 | Optional Arguments | 23 |
| 3.4 | What's Next | 24 |
| 4. | Manipulating the DOM | 25 |
| 4.1 | dart:html | 25 |
| 4.2 | Finding Things | 25 |
| 4.3 | Adding Things | 27 |

| | | |
|--|--|----|
| 4.4 | Removing Things | 28 |
| 4.5 | Updating Elements | 29 |
| 4.6 | DOM Ready | 30 |
| 4.7 | What's Next | 30 |
| 5. | Compiling to JavaScript | 31 |
| 5.1 | Compiling to JavaScript with dart2js | 32 |
| 5.2 | Maintaining Dart and JavaScript Side-by-Side | 34 |
| 5.3 | What's Next | 36 |
| Part II — Effective Coding Techniques | | |
| 6. | Project: MVC in Dart | 39 |
| 6.1 | MVC in Dart | 39 |
| 6.2 | Hipster Collections | 41 |
| 6.3 | Hipster Models | 44 |
| 6.4 | Hipster Views | 48 |
| 6.5 | Working with MVC to Delete | 51 |
| 6.6 | What's Next | 52 |
| 7. | Classes and Objects | 53 |
| 7.1 | Class Is a First-Order Concept | 53 |
| 7.2 | Instance Variables | 53 |
| 7.3 | Methods | 55 |
| 7.4 | Static Methods and Variables (aka Class Methods and Variables) | 58 |
| 7.5 | Interfaces | 59 |
| 7.6 | Subclasses | 60 |
| 7.7 | Constructors | 61 |
| 7.8 | What's Next | 66 |
| 8. | Events | 67 |
| 8.1 | Plain-Old Events | 67 |
| 8.2 | Custom Event Systems | 68 |
| 8.3 | What's Next | 71 |

Part III — Code Organization

| | | |
|------|--|----|
| 9. | <u>Project: Extracting Libraries</u> | 75 |
| 9.1 | <u>What to Extract and What to Leave</u> | 75 |
| 9.2 | <u>Real Libraries</u> | 80 |
| 9.3 | <u>What's Next</u> | 82 |
| 10. | <u>Libraries</u> | 83 |
| 10.1 | <u>#source()</u> | 83 |
| 10.2 | <u>#import()</u> | 85 |
| 10.3 | <u>Core Dart Libraries</u> | 87 |
| 10.4 | <u>What's Next</u> | 87 |

Part IV — Maintainability

| | | |
|------|--|-----|
| 11. | <u>Project: Varying Behavior</u> | 91 |
| 11.1 | <u>Vary Class Behavior with noSuchMethod()</u> | 91 |
| 11.2 | <u>Sync Through Dependency Injection</u> | 95 |
| 11.3 | <u>What's Next</u> | 99 |
| 12. | <u>Testing Dart</u> | 101 |
| 12.1 | <u>Obtaining the Test Harness</u> | 101 |
| 12.2 | <u>2 + 2 = 5 Should Be Red</u> | 101 |
| 12.3 | <u>What's Next</u> | 106 |

Part V — The Next Level with Dart

| | | |
|------|---|-----|
| 13. | <u>Project: An End to Callback Hell</u> | 109 |
| 13.1 | <u>The Future</u> | 109 |
| 13.2 | <u>Handling Errors in the Future</u> | 112 |
| 13.3 | <u>What's Next</u> | 113 |
| 14. | <u>Futures and Isolates</u> | 115 |
| 14.1 | <u>Completers and Futures</u> | 115 |
| 14.2 | <u>Isolates</u> | 116 |
| 14.3 | <u>Heavy vs. Light Isolates</u> | 119 |
| 14.4 | <u>Wrapping Up</u> | 119 |
| 15. | <u>HTML5 and Dart</u> | 121 |
| 15.1 | <u>Animation</u> | 121 |
| 15.2 | <u>Local Storage</u> | 122 |

| | | |
|------|------------------------------------|-----|
| 15.3 | <u>WebSockets</u> | 124 |
| 15.4 | <u>Canvas</u> | 126 |
| 15.5 | <u>Wrapping Up</u> | 128 |

Introduction

Why Dart?

When I ask that question, I do not wonder why Google is pursuing Dart. I am not asking what the language designers are hoping to accomplish. To be sure, we will touch on both those questions and many more in the course of this book.

When I ask “Why Dart?” it is a question meant for myself. What on Earth makes me think this is a good language to learn, let alone write an entire book about? Especially at the 0.08 release.

The answer to that question is a personal and professional journey to understand how to make the Internet as fast as possible. Back in the day, I was a simple Perl hacker. I rather liked the language and what I could do with it. But when Ruby and Ruby on Rails hit the scene, I jumped. The combination of simple, clean code and strong convention won me over. For a while.

Next, I explored smaller frameworks like Sinatra,¹ which retain the beauty of the Ruby language but lead to smaller and faster code. The two frameworks sufficed as a nice continuum for web development, and I was happy.

But perhaps there was more? This eventually led me to Node.js and various JavaScript frameworks built on top of it. And it seemed that no more speed could possibly be eked out of servers.

And then I found the SPDY² protocol, which fascinated me so much that I wrote *The SPDY Book*.³ Here, at last, was an attempt not just to improve on what we have but to redefine the rules of the game.

1. <http://sinatrarb.com>

2. <http://www.chromium.org/spdy>

3. <http://spdybook.com/>

One of the things that I noticed while working with SPDY was that no matter how much I took advantage of what the protocol offered, the ultimate speed of web applications was limited by how quickly the web page and client-side scripts, CSS, and so on, could be evaluated.

JavaScript has been around for seventeen years. When it was first introduced, there was no Web 2.0, no Ajax, no CSS, and very little client-side interactivity at all. When JavaScript first came out, the primary use case was validating client-side forms with alert boxes!

Over the next seventeen years, JavaScript the language has evolved from a proprietary language owned and slowly developed by Netscape Communications to a web standard that regularly adds new features. But as fast as a committee can add new features to the standard, the Web evolves infinitely faster.

And then along came Dart. Dart asks, given what we know about the Web today, how might we build JavaScript from scratch? How can it load and run as fast as possible? How can we write it so that we can easily define and load external libraries?

How can we make it easy for developers to write beautiful code?

If Dart is the answer to those many questions (and I will try to make the case that it is), then Dart is quite possibly the most exciting technology to come our way in a very long time.

And that is the answer to “Why Dart?”

Who Should Read This Book (Besides Hipsters)?

I am writing this book primarily for any developer looking to keep their JavaScript skills as fresh as possible. The best way to improve JavaScript skills is through practice and reading other people’s code. But sometimes it can radically help to see what the competition is up to. In this case, as we explore what Dart brings to the table, we can better understand the gaps in an admittedly wonderful language.

I also hope that this book will prove useful for the newly converted. Dart is already a worthy platform for building insanely fast web applications for today’s web browsers. I hope that after reading this book, you will be well armed to produce the next generation of web applications.

This book should be of interest to developers learning languages for the sake of learning. I focus quite a bit on the Dart language, especially in those places that it surprised and delighted me.

And of course, hipsters should read as well. Dart is just different enough to make it intriguing to the typical language hipster and yet powerful enough to make it worthwhile for the hipster who hopes to change the world.

How the Book Is Organized

I am trying something different with this book. Rather than introducing slices of the language in each chapter, I bite off chunks. Each section starts with an actual Dart project, including some commentary on the choices being made. My goal in these sections is to leverage Dart's avowed familiarity to make significant headway in giving a real feel for what the language is. Since these are real projects, they are great opportunities to point out Dart's strengths and, yes, some of its weaknesses.

Each of these project chapters is followed by smaller topic-specific chapters that go into a bit more depth about aspects of the language. I use these to cover material that is too detailed for the project chapters and material that cannot be found in current Dart reference material.

So, if you want a quick introduction to the language, you can certainly start by reading the project chapters alone. If you want a more traditional book, then skip the project chapters and read just the topic chapters. Or read it all—I will try to make it worth your time!

The first project is [Chapter 1, *Project: Your First Dart Application*, on page 3](#). Supplementing that project are [Chapter 2, *Basics Types*, on page 11](#); [Chapter 3, *Functional Programming in Dart*, on page 19](#); [Chapter 4, *Manipulating the DOM*, on page 25](#); and [Chapter 5, *Compiling to JavaScript*, on page 31](#).

The next project is taking the simple Ajax application from [Chapter 1, *Project: Your First Dart Application*, on page 3](#) and whipping it into a full-fledged MVC library in [Chapter 9, *Project: Extracting Libraries*, on page 75](#). If you want to put a language through its paces, writing a library, especially an MVC library, is a great way to do it. Following up on the MVC library, we have [Chapter 7, *Classes and Objects*, on page 53](#); [Chapter 10, *Libraries*, on page 83](#); and [Chapter 8, *Events*, on page 67](#).

Following that, we will take a look at dependency injection in Dart with [Chapter 11, *Project: Varying Behavior*, on page 91](#). Unlike JavaScript, Dart is not primarily a dynamic language, though as we see in that chapter, it is still

possible to perform some tricks of traditional dynamic languages. The follow-up to that project is an introduction to Dart testing, which is an important topic, even if not quite baked in Dart.

The last project chapter is [Chapter 13, *Project: An End to Callback Hell*, on page 109](#), in which we explore Dart “futures” as a higher-order replacement for traditional callback passing. This leads into a discussion of code isolation and message passing in [Chapter 14, *Futures and Isolates*, on page 115](#).

Finally, we conclude the book with a brief exploration of various HTML5 technologies that are not covered elsewhere in the book.

What Is Not in This Book

We will not cover the Dart Editor. In some regards, this is something of a loss—strongly typed languages like Dart lend themselves to code completion, of which the Dart Editor takes advantage. Still, the focus of the book is meant to be the language, not the tools built around it. Besides, some people (myself included) will want to stick with their code editor of choice.

This book is not intended as a language reference. It is too early in the evolution of this language for a reference. Still, the hope is that this book will prove a strong supplement for the specification (which is not meant to be developer-friendly)⁴ and the API documentation (which is still incomplete in places).⁵

About the Future

Since Dart will continue to evolve, so will this book. Once or twice a year, depending on how quickly Dart changes, the content in this book will be reviewed and then revised, removed, or supplemented.

If you identify any mistakes or areas in need of improvement, please record them in the public issue tracker: <https://github.com/dart4hipsters/dart4hipsters.github.com/issues>. Suggestions for new topics to cover are also welcome!

Conventions

Class names are camel-cased (for example, `HipsterModel`). Classes have filenames that are identical to the class names (for example, `HipsterModel.dart`). Variable

4. The Dart specification is kept at <http://www.dartlang.org/docs/spec/>. It is intended for the language implementers but can be useful for application developers in a pinch.

5. <http://api.dartlang.org>

names are snake-cased (for example, `background_color`), while functions and methods are lowercase camel-cased (for example, `routeToRegExp()`).

Let's Get Started

With the preliminaries out of the way, let's get started coding for the Web without the legacy of the Web. Let's code some Dart!

Part I

Getting Started

Dart has a lot going for it, but perhaps the most impressive feature is how familiar it is to programmers familiar with JavaScript. In these first few chapters, with no previous experience, we will write a Dart application.

Project: Your First Dart Application

Most programming books start with a “Hello World!” sample. I say, screw that—we’re all hipsters here. Let’s start coding!

Since Dart is written, above all else, to be familiar, we should not be too far out of our depths diving right in. Let’s jump straight to something more fun: an Ajax-powered website. Any true hipster has an extensive collection of comic books (am I right? I’m not the only one, am I?), so let’s consider a simple Dart application that manipulates the contents of that collection via a REST-like interface.

At some point, this may prove too much of a whirlwind. Have no fear, we will go into details in subsequent chapters.

1.1 The Back End

Sample code for this chapter can be found in the “your_first_dart_app” branch of <https://github.com/eee-c/dart-comics>. As hipsters, we’re already using Node.js, so the back end requires Node.js and npm. Instructions are contained in the project’s README.

Being REST-like, the application should support the following:

- GET /comics (return a list of comic books)
- GET /comics/42 (return a single comic book)
- PUT /comics/42 (update a comic book entry)
- POST /comics (create a new comic book in the collection)
- DELETE /comics/42 (delete a comic book)

We will not worry too much about the details of the back end beyond that.

1.2 HTML for Dart

Our entire application will follow the grand tradition of recent client-side MVC frameworks. As such, we require only a single web page.

`your_first_dart_app/index.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Dart Comics</title>
  <link rel="stylesheet" href="/stylesheets/style.css">

  <!-- Force Dartium to start the script engine -->
  <script language="text/javascript">
    navigator.webkitStartDart();
  </script>

  <!-- The main application script -->
  <script src="/scripts/comics.dart"
    type="application/dart"></script>
</head>

<body>
  <h1>Dart Comics</h1>
  <p>Welcome to Dart Comics</p>

  <ul id="comics-list"></ul>

  <p id="add-comic">
    Add a sweet comic to the collection.
  </p>
</body>
</html>
```

Most of that web page should be familiar; it will include simple HTML, links for CSS, and scripts.

HTML Head

The only oddity to note is the first `<script>` tag, in which *JavaScript* starts the Dart scripting engine.

`your_first_dart_app/_index_force_dartium_script_engine.html`

```
<!-- Force Dartium to start the script engine -->
<script language="text/javascript">
  navigator.webkitStartDart();
</script>
```

Important: At the time of this writing, it is necessary to kick-start the Dart VM with `navigator.webkitStartDart()` on Dartium, the Dart-enabled version of Chrome.¹ This requirement should go away in the very near future.

Next we load the contents of our actual code. The only change here is a different type attribute in the `<script>` tag, indicating that this is Dart code.

```
your_first_dart_app/_index_src_dart.html
<!-- The main application script -->
<script src="/scripts/comics.dart"
        type="application/dart"></script>
```

There is more to be said about loading libraries and including code with Dart once we reach [Chapter 10, Libraries, on page 83](#). For now, it is simply nice to note that loading Dart works exactly as we might expect it to work.

HTML Body

As for the body of the HTML, there is nothing new there, but we ought to note the IDs of two elements to which we will be attaching behaviors.

```
your_first_dart_app/_index_body.html
<h1>Dart Comics</h1>
<p>Welcome to Dart Comics</p>
<ul id="comics-list"></ul>
<p id="add-comic">
  Add a sweet comic to the collection.
</p>
```

To the `#comics-list` UL element, we are going to attach the list of comic books in the back-end data store. We will also attach a form handler to the `#add-comic` paragraph tag. So, let's get started.

1.3 Ajax in Dart

We start our Dart application by loading a couple of Dart libraries with a `main()` function in `scripts/comics.dart`.

```
your_first_dart_app/comics_initial_main.dart
#import('dart:html');
#import('dart:json');
main() {
  load_comics();
}
load_comics() {
  // Do stuff here
}
```

1. <http://www.dartlang.org/dartium/>

As we will see in [Chapter 10, Libraries, on page 83](#), there is a lot of power in those `#import` statements. For now, we can simply think of them as a means for pulling in functionality outside of the core Dart behavior.

All Dart applications use `main()` as the entry point for execution. Simply writing code and expecting it to run, as we do in JavaScript, will not work here. It might seem C-like at first, but does it honestly make sense that code lines strewn across any number of source files and HTML will all start executing immediately? The `main()` entry point is more than convention; it is a best practice enforced by the language.

As for the `load_comics()` function, we take it piece by piece. We need to identify the DOM element to which the list will attach (`#comics-list`). Next we need an Ajax call to fill in that DOM element. To accomplish both of those things, our first bit of Dart code might look like the following:

```
your_first_dart_app/_load_comics.dart
load_comics() {
  var list_el = document.query('#comics-list');
  ajax_populate_list(list_el);
}
```

Aside from the obvious omission of the function keyword, this example might be JavaScript code! We will cover more differences in [Chapter 3, Functional Programming in Dart, on page 19](#). Still in Dart are the semicolons and curly braces that we know and love—the language designers have certainly made the language at least superficially familiar.

Note: Unlike in JavaScript, semicolons are *not* optional in Dart.

In addition to being familiar, this code is easy to read and understand at a glance. There are no weird, legacy DOM methods. We use `document.query()` for an element rather than `document.getElementById()`. And we use the familiar CSS selector of `#comics-list`, just as we have grown accustomed to in jQuery.

Having found the UL we want to populate, let's see how to make an Ajax request. As in JavaScript, we create a new XHR object, add a handler for when the request loads, and then open and send the request.

```
your_first_dart_app/_ajax_populate_list.dart
ajax_populate_list(container) {
  var req = new XMLHttpRequest();

  req.onload.add((event) {
    var list = JSON.parse(req.responseText);
    container.innerHTML = graphicNovelsTemplate(list);
  });
}
```

```
// verb, resource, boolean async
req.open('get', '/comics', true);
req.send();
}
```

Most of that code should be immediately familiar to anyone who has done Ajax coding in the past. We open by creating an XHR object and close by specifying the resource to be retrieved and actually sending the request.

It's when we add event handlers that we see a departure from the JavaScript way. The XHR object has an `on` property that lists all supported event handlers. We access one of those handler types, `load`, so that we can add a handler to it, with the appropriately named `add()` method. In this case, we parse the supplied JSON into a list of hashes, which might look like this:

`your_first_dart_app/comics.json`

```
[
  {"title": "Watchmen",
   "author": "Alan Moore",
   "id": 1},
  {"title": "V for Vendetta",
   "author": "Alan Moore",
   "id": 2},
  {"title": "Sandman",
   "author": "Neil Gaiman",
   "id": 3}
]
```

With that, we hit the final piece of our simple Dart application—a template for populating the list of comic books.

`your_first_dart_app/_graphic_novels_template.dart`

```
graphic_novels_template(list) {
  var html = '';
  list.forEach((graphic_novel) {
    html += _graphic_novel_template(graphic_novel);
  });
  return html;
}

_graphic_novel_template(graphic_novel) {
  return """
    <li id="${graphic_novel['id']}">
      ${graphic_novel['title']}
      by
      ${graphic_novel['author']}
    </li>
    """;
}
```

The first function simply iterates over our list of comic books (internally, we think of them as graphic novels), building up an HTML string.

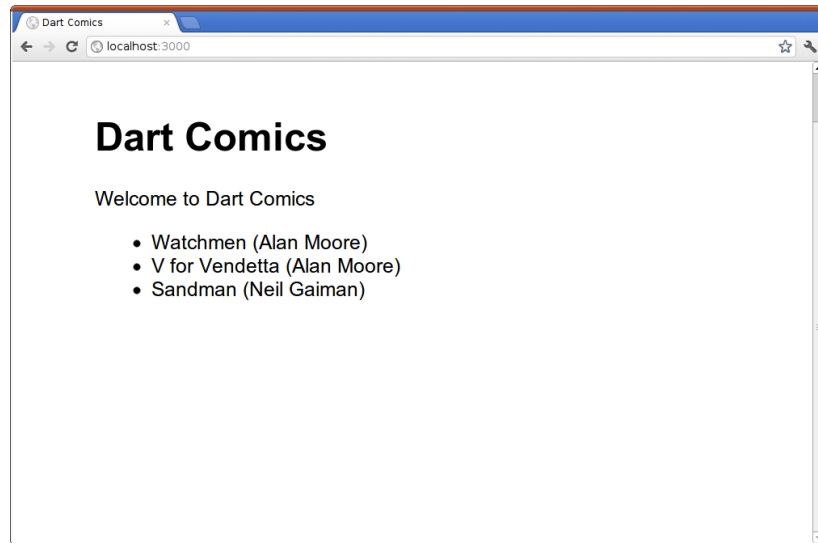
The second function demonstrates two other Dart features: multiline strings and variable interpolation. Multiline strings are identified by three quotes (single or double). Inside the string, we can interpolate values (or even simple expressions) with a dollar sign. For simple variable interpolation, curly braces are optional: `$name` is the same as `${name}`. For more complex interpolation, such as hash lookup, the curly braces are required.

And that's it! We have a fully functional, Ajax-powered web application ready to roll. The assembled code is as follows:

`your_first_dart_app/comics.dart`

```
#import('dart:html');
#import('dart:json');
main() {
  load_comics();
}
load_comics() {
  var list_el = document.query('#comics-list');
  ajax_populate_list(list_el);
}
ajax_populate_list(container) {
  var req = new XMLHttpRequest();
  req.on.load.add((event) {
    var list = JSON.parse(req.responseText);
    container.innerHTML = graphicNovelsTemplate(list);
  });
  // verb, resource, boolean async
  req.open('get', '/comics', true);
  req.send();
}
graphic_novels_template(list) {
  var html = '';
  list.forEach((graphic_novel) {
    html += _graphic_novel_template(graphic_novel);
  });
  return html;
}
_graphic_novel_template(graphic_novel) {
  return """
    <li id="${graphic_novel['id']}">
      ${graphic_novel['title']}
      by
      ${graphic_novel['author']}
    </li>
  """;
}
```

And loading the page looks like this:



That is a darned nice start in our exploration of Dart. To be sure, we glossed over a lot of what makes Dart a great little language. But in doing so, we have ourselves a very good start on an Ajax-powered web application. Best of all, none of the code that we wrote seemed all that different from JavaScript. Some of the syntax is a little cleaner than what we are used to in JavaScript (no one is going to complain about cleaner code), and those strings are quite nice. But, all in all, it is safe to say that we can be productive with Dart in relatively short order.

1.4 This App Won't Run

As written, this application will not actually work anywhere...well, almost anywhere.

Dart is not supported in any browser (not even Chrome). To run this web application natively, we would need to install Dartium—a branch of Chrome that embeds the Dart VM. Dartium is available from the Dart Lang site.²

Even after Dart makes it into Chrome proper, we would still be faced with supporting only a subset of browsers on the market. That is just silly.

Fortunately, Dart can be compiled down to JavaScript, meaning that you can have the power of Dart but still target all platforms. To accomplish that easily,

2. <http://www.dartlang.org/dartium/>

we add a small JavaScript library that, upon detecting a browser that does not support Dart, will load the compiled JavaScript equivalent.

```
your_first_dart_app/_index_src_js_fallback.html
```

```
<!-- Enable fallback to Javascript -->  
<script src="/scripts/conditional-dart.js"></script>
```

We will discuss that helper file in detail in [Chapter 5, *Compiling to JavaScript*, on page 31](#). For now, it is enough to note that our Dart code is not locked into a single browser vendor's world. We are very definitely *not* seeing *The Return of VBScript* here.

1.5 What's Next

Admittedly, this has been a whirlwind of an introduction to Dart. It is fantastic to be able to get up and running this quickly. It is even better to feel as though we can be productive at this point.

Still, we are only getting started with Dart, and, make no mistake, our Dart code can be improved. So, let's use the next few chapters to get comfortable with some important concepts in Dart. After that, we will be ready to convert our Dart application into an MVC approach in [Chapter 6, *Project: MVC in Dart*, on page 39](#).

Basics Types

A recurring theme in this book is that Dart aims to be familiar. If that holds true, then a discourse on basic components of the language should be relatively brief—and it will be. Even so, some introduction to core types can only help. And, naturally, there are a few “gotchas” here and there.

2.1 Numbers

Integers and doubles are both number types, which means that both support many of the same methods and operators. Dart numbers work pretty much like they do in many other languages.

```
2 + 2;      // 4
2.2 + 2;    // 4.2
2 + 2.2;    // 4.2
2.2 + 2.2;  // 4.4
```

As can be seen, Dart numbers do “the right thing” when mixing them in operations.

2.2 Strings

Strings are immutable, which is a fancy way of saying that string operations create new strings instead of modifying existing strings. Strings (like numbers) are hashable, meaning that unique objects have unique hash codes to tell them apart. If we assign a variable to a variable holding a string, both will have the hash code because they are the same object.

```
var str1 = "foo",
    str2 = str1;

str1.hashCode(); // 425588957
str2.hashCode(); // 425588957
```


But, if we modify the first string, the result will be an entirely new object while the copy continues to point to the original string.

```
str1 += "bar";

str1.hashCode(); // 447120306
str2.hashCode(); // 425588957
```

Dart goes out of its way to make working with strings easy. It is possible to create multiline strings by enclosing them in triple quotes.

```
"""Line #1
Line #2
Line #3""";
```

Dart also considers adjacent strings to be concatenated.

```
'foo' ' ' 'bar'; // 'foo bar'
```

This adjacent string convenience even extends to multiline strings.

```
'foo'
' '
'bar'; // 'foo bar'
```

Important: This adjacent string concatenation does not work everywhere at the time of this writing but is definitely part of the planned specification.

A last convenience of Dart strings is variable interpolation. Dart uses \$ to denote variables to be interpolated.

```
var name = "Bob";

"Howdy, $name"; // "Howdy, Bob"
```

If there is potential for confusion over where the variable expression ends and the string begins, curly braces can be used with \$.

```
var comic_book = new ComicBook("Sandman");

"The very excellent ${comic_book.title}!";
// "The very excellent Sandman"
```

2.3 Booleans

The values true and false are the only allowed boolean (bool) type in Dart. The notion of “truthiness” does not get simpler than it is in Dart: if it’s not true, then it’s false. Consider the following:

```

var name, greeting;

greeting = name ? "Howdy $name" : "Howdy";
// "Howdy"

/** Name is still not true */
name = "Bob";
greeting = name ? "Howdy $name" : "Howdy";
// "Howdy"

greeting = (name != null) ? "Howdy $name" : "Howdy";
// "Howdy Bob"

```

If you are coming from many other languages, then it will not be a surprise that `null`, `""`, and `0` evaluate to `false` in a boolean context. It may take some getting used to `"Bob"` and `42` evaluating to `false` as well.

The semantics for truthiness vary slightly in “type-checked” mode (described in [Section 2.7, Types, on page 17](#)), but it is best not to rely on such minor variations. If we always assume that the previous will hold, then we will not get burned.

Warning: In some older implementations of Dart, the second greeting in the previous code (`name ? "Howdy $name" : "Howdy"`) *would* evaluate `name` as `true`, resulting in `"Howdy Bob"`. Do not rely on this behavior because it will change to yield the exact opposite of what you might expect.

In [Chapter 7, Classes and Objects, on page 53](#), we will explore operator definition, which allows class-specific definitions of `equals` / `==`. This gives Dart a certain amount of flexibility with regard to booleans.

2.4 HashMaps (aka Hashes, Associative Arrays)

Key-value pairs are implemented in Dart by `HashMap` objects. Defining an options hash creates a `HashMap`.

```

var options = {
  'color': 'red',
  'number': 2
};

```

As you’d expect, retrieving values from a `HashMap` is done with square brackets.

```

var options = {
  'color': 'red',
  'number': 2
};
options['number']; // 2

```

HashMap implements the Map interface, which is where most of the relevant API documentation can be found.¹ This includes information on retrieving keys (getKeys()) and values (getValues()) and iterating over the entire object with forEach().

```
var options = {
  'color': 'red',
  'number': 2
};

options.forEach((k, v) {
  print("$k: $v");
});
// number: 2
// color: red
```

Note: The order of key-value pairs is not guaranteed.

One *extremely* useful feature of HashMap is the putIfAbsent() method. The following two are equivalent:

```
// weak
if (!options.containsKey('age')) {
  var dob = new Date.fromString('2000-01-01'),
      now = new Date.now();
  options['age'] = now.year - dob.year;
}

// confident
options.putIfAbsent('age', findAge);

findAge() {
  var dob = new Date.fromString('2000-01-01'),
      now = new Date.now();
  return now.year - dob.year;
};
```

In the first example, both the conditional and the block had to concern themselves with implementation details of the options HashMap. In the second example, the findAge() function is concerned solely with calculating the current age, while the options HashMap worries only about adding the value.

Important: Always seek opportunities to use putIfAbsent(). Your Dart will be much cleaner for it.

The first example could be cleaned up to be the following:

```
if (!options.containsKey('age')) {
  options['age'] = findAge();
}
```

1. http://api.dartlang.org/dart_core/Map.html

But, without `putIfAbsent()`, it is likely that we would have left the `findAge` implementation inside the conditional. Regardless, the conditional format is never going to be as clean as the `putIfAbsent()` equivalent.

```
options.putIfAbsent('age', findAge);
```

`putIfAbsent()`—learn it, love it. It will save your life (well, probably not, but it'll make life that much sweeter).

2.5 Lists (aka Arrays)

Lists of things are a requirement of any language. Easing developers into the language, Dart sticks close to the expected with lists.

```
var muppets = ['Count', 'Bert', 'Snuffleupagus'];
var primes = [1, 2, 3, 5, 7, 11];
// Indexed from zero
muppets[0];    // 'Count'
primes.length; // 6
```

Dart does provide some nice, *consistent* methods for manipulating lists.

```
var muppets = ['Count', 'Bert', 'Ernie', 'Snuffleupagus'];

muppets.setRange(1, 2, ['Kermit', 'Oscar']);
// muppets => ['Count', 'Kermit', 'Oscar', 'Snuffleupagus']

muppets.removeRange(1, 2);
// muppets => ['Count', 'Snuffleupagus'];

muppets.addAll(['Elmo', 'Cookie Monster']);
// muppets => ['Count', 'Snuffleupagus', 'Elmo', 'Cookie Monster']
```

There are a number of iterating methods built-in as well.

```
var muppets = ['Count', 'Bert', 'Ernie', 'Snuffleupagus'];

muppets.forEach((muppet) {
  print("$muppet is a muppet.");
});
// =>
// Count is a muppet.
// Bert is a muppet.
// Ernie is a muppet.
// Snuffleupagus is a muppet.

muppets.some((muppet) {
  return muppet.startsWith('C');
});
// true
```

```

muppets.every((muppet) {
  return muppet.startsWith('C');
});
// false

muppets.filter((muppet) {
  return muppet.startsWith('C');
});
// ['Count']

```

Important: As of this writing, Dart lacks a `reduce()` or `fold()` method for performing higher-order operations on lists.

Thankfully, there is not much that needs to be introduced for Dart lists and arrays. They are one of many things in Dart that “just work.”

Collections

The iterating methods from the previous section are not actually defined on the `List` class. Rather, they come from `List`’s superclass: `Collection`. Two other members of the `Collection` family are `Set` and `Queue`.

The `Set` class is a `List` in which the elements are always unique and that exposes some set operations.

```

var sesame = new Set.from(['Kermit', 'Bert', 'Ernie']);
var muppets = new Set.from(['Piggy', 'Kermit']);

// No effect b/c Ernie is already in the Set
sesame.add('Ernie');           // => ['Kermit', 'Bert', 'Ernie']
sesame.intersection(muppets); // => ['Kermit']
sesame.isSubsetOf(muppets);   // => false

```

The `Queue` is a `List` that can be manipulated at the beginning.

```

var muppets = new Queue.from(['Piggy', 'Rolf']);

muppets.addFirst('Kermit');
// muppets => ['Kermit', 'Piggy', 'Rolf']

muppets.removeFirst();
muppets.removeLast();
// muppets => ['Piggy']

```

The corollary to the existence of `Queue` is that regular lists cannot be manipulated at the beginning. That is, there is no `shift` or `unshift` method for `List`.

2.6 Dates

Dart brings some much needed sanity to dates and times in the browser. The answer to the question that is burning in many a JavaScript refugee’s heart is “yes”—the first month is, in fact, 1. Let the rejoicing commence.

The niceties of Dart dates do not end there. For instance, there are a number of ways to create dates.

```
var mar = new Date.fromString('2012-03-01 Z-0500');
// 2012-03-01 00:00:00.000

var now = new Date.now();
// 2012-03-10 01:02:24.149

var apr = new Date(2012, 4, 1, 0, 0, 0, 0);
// 2012-04-01 00:00:00.000
```

Even better, manipulating dates is not only possible but quite nice.

```
var mar = new Date(2012, 3, 1, 0, 0, 0, 0);
var apr = new Date(2012, 4, 1, 0, 0, 0, 0);

var diff = apr.difference(mar);
diff.inDays; // => 31

apr.add(new Duration(days: 15)); // => 2012-04-16
```

The `difference()` method in `Date` returns a `Duration` object that encapsulates a period of time. A `Duration` can be queried in any number of time units—from days all the way down to milliseconds. As we can see with our `add()` example, `Durations` also come in handy when adding or removing time from a particular date.

Working with dates in Dart is not a thing to dread. As can be seen already, they are downright pleasant.

2.7 Types

Important: It is *highly* recommended that coding Dart is done in type-checked mode. By default, Dart runs in “production” mode, which will not crash when presented with seemingly conflicting type definitions. It is therefore incumbent upon the developer to catch as many such problems as possible in development mode. To enable type-checked mode, start Dartium from the command line with `DART_FLAGS='--enable_type_checks --enable_asserts' /path/to/dartium`.

Before moving on to other topics, let’s take a quick look at declaring variables in Dart. So far, we have been following the JavaScript convention of declaring

them with the `var` keyword. In Dart, `var` indicates a variable type. In other words, not only are we not specifying a type, but we are telling the interpreter that the type can change.

```
var muppet = 'Piggy';

// Dart, like JavaScript, allows this, but come on!
muppet = 1;
```

Dart is a somewhat strongly typed language, meaning that it prefers that we declare types instead of `var`.

```
String muppet = 'Piggy';

// Fails in type-check mode
// Complains otherwise
muppet = 1;
```

Dart will still allow us to do something silly as shown in the previous code, but it will complain. In type-checked mode, which can be enabled at the command line, the previous code will throw an exception.

Although the `var` keyword is acceptable, it is generally considered good manners to declare types.

```
int i = 0;
bool is_done = false;
String muppet = 'Piggy';
Date now = new Date.now();
```

For types that contain other types, it is also possible to declare the type of the objects being stored.

```
HashMap<String,bool> is_awesome = {
  'Scooter': false,
  'Bert': true,
  'Ernie': false
};

List<int> primes = [1,2,3,5,7,11];
```

Declaring types makes intent easier to read, which is important for maintainability. Types will also aid the interpreter when compiling your code, allowing it to run faster.

2.8 What's Next

There was a lot jammed into this chapter. Much of Dart should still feel familiar at this point, with some key but (I hope) welcome differences.

Functional Programming in Dart

Some of what makes JavaScript special is its support for functional programming. Since Dart aims to be familiar, let's examine what it is like to program functionally in Dart.

We begin with the old standby, the Fibonacci sequence. In JavaScript, this might be written like so:

```
function fib(i) {  
  if (i < 2) return i;  
  return fib(i-2) + fib(i-1);  
}
```

The Fibonacci sequence is a wonderful example for exploring the functional programming nature of a language since it, er, is a function but also because it demonstrates how to invoke a function because of its recursive nature.

I won't bother describing recursion or the particulars of this function.¹ Instead, let's focus on how to use the function in JavaScript.

```
fib(1) // => 1  
fib(3) // => 3  
fib(10) // => 55
```

So, JavaScript functions are simple enough. They are introduced with the function keyword, followed by the name of the function, the list of supported arguments in parentheses, and the block that describes the body of the function.

So, what would the equivalent Dart version look like?

1. The Fibonacci sequence is well-documented elsewhere if you need a refresher: http://en.wikipedia.org/wiki/Fibonacci_number.


```
// Dart
fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

Wait, how is that different from the JavaScript version?

```
function fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

Astute readers will note that the Dart version lacks the function keyword. Aside from that, the two function definitions are identical, as is invoking the two.

```
fib(1); // => 1
fib(5); // => 5
fib(10); // => 55
```

If nothing else, we can see that the designers of the Dart language have certainly succeeded in producing something familiar.

3.1 Anonymous Functions

The experienced JavaScript programmer is well-versed in using anonymous functions. Since functions in JavaScript are a first-order concept, functions are passed around in JavaScript with abandon. Some even lament the callback hell of certain frameworks, but aesthetics aside, there can be no denying that anonymous functions are an important thing in JavaScript. So, the same must surely be true in Dart, right?

In JavaScript, an anonymous function omits the function name, using only the function keyword.

```
function(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

We have already seen that the only difference between JavaScript and Dart functions is that the latter do not have the function keyword. It turns out this is also the only difference between JavaScript and Dart anonymous functions.

```
(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

At first glance, that looks quite odd—almost naked. But that is just our JavaScript eye. Ruby has lambdas and procs that look very similar.

```
{ |i| $stderr.puts i }
```

Given enough consideration, what purpose does the function keyword in JavaScript really serve? The knee-jerk reaction is that it helps to identify the anonymous function, but in practice, it is just noise.

Consider this Fibonacci printer:

```
var list = [1, 5, 8, 10];
list.forEach(function(i) {fib_printer(i)});

function fib_printer(i) {
  console.log("Fib(" + i + "): " + fib(i));
}

function fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

Does the function keyword help or hinder readability of the code? Clearly, it makes the situation worse, especially inside the `forEach()` call.

Let's consider the equivalent Dart code.

```
var list = [1, 5, 8, 10];
list.forEach((i) {fib_printer(i)});

fib_printer(i) {
  print("Fib($i): " + fib(i));
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

Note: We are using the string interpolation trick first noted in [Chapter 1, Project: Your First Dart Application, on page 3](#) to insert the value of the counter `i` into `"Fib($i)"`.

All that we did was to remove the function keyword, and yet the intent of the code is much clearer. Multiply that effect across an entire project, and the long-term health of a codebase goes up dramatically.

Speaking of clarity, if curly braces make you cringe, there is a hash rocket syntax that can be used for *simple* functions. Instead of writing our anonymous

iterator as `(i) { fib_printer(i) }`, we can write `(i) => fib_printer(i)`. Thus, our code becomes as follows:

```
var list = [1, 5, 8, 10];
list.forEach((i) => fib_printer(i));

fib_printer(i) {
  print("Fib($i): " + fib(i));
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

The argument `(i)` is repeated both in the definition of the anonymous function and in the call to `fib_printer(i)`. In JavaScript, there is nothing to be done to clean that up. In Dart, however, the function `(i) => fib_printer(i)` can be further simplified as simply `fib_printer`.

```
var list = [1, 5, 8, 10];
list.forEach(fib_printer);

fib_printer(i) {
  print("Fib($i): " + fib(i));
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

That is a fantastic little shortcut to use with abandon in our Dart code.

3.2 First-Order Functions

Passing an anonymous function into an iterator like `forEach()` already demonstrates some nice support for first-class objects—the ability to treat functions as variables that can be assigned and passed around.

At the time of this writing, Dart lacks facilities (for example, reflection) to support sophisticated functional concepts such as currying or combinators. That said, it is already possible to perform partial function application in Dart.

The classic example of partial application is converting an `add()` function that returns the sum of three numbers into another function that fixes one of those numbers.

```

add(x, y, z) {
  return x + y + z;
}

makeAdder2(fn, arg1) {
  return (y, z) {
    return fn(arg1, y, z);
  };
}

var add10 = makeAdder2(add, 10);

```

The name *partial application* comes from returning a function with one argument already applied. In this case, the `makeAdder2` function returns another function taking two arguments. The result of calling this new function is the same as calling the original function with the first argument fixed to `arg1`.

At this point, the `add10()` function takes two numbers, sums them, and ups the total by ten.

```
add10(1,1); // => 12
```

Unfortunately, this is the limit of Dart combinators at this time. That will change shortly because reflection is being actively worked on.

3.3 Optional Arguments

One of the more tedious things to do in JavaScript applications is extracting optional arguments. This is solved in Dart with a built-in syntax that encapsulates this concept.

In the following, the optional arguments are those enclosed in square brackets:

```

f(a, [b1='who', b2, b3, b4, b5, b6, b7]) {
  // ...
}

```

It is possible to invoke this function without any of the optional arguments with this form: `f('foo')`. In this case, the parameter `a` inside the function body would be assigned `'foo'`.

To assign optional parameters, prefix them with the parameter name in the function call.

```
f('foo', b6:'bar', b3:'baz');
```

The result of calling the previous function would be that, inside `f()`, the variable `a` would be assigned `'foo'`, `b1` would be `'who'`, `b3` would be `'baz'`, and `b6` would be `'bar'`. All of the remaining optional arguments, `b2`, `b4`, `b5`, and `b7`, would be `null`.

Of particular note here is that we can specify default values for optional arguments with an assignment in the function parameter list. In this case, we have defaulted the value of the `b1` variable to the string `'who'`.

This certainly beats rooting through object literals. Optional arguments are even more powerful inside class and instance methods, as we will see in [Chapter 7, *Classes and Objects*, on page 53](#).

A Quick Word About this: One of the ways in which Dart departs from JavaScript is in the use of the `this` keyword. Dart's stance on the matter is that `this` has absolutely nothing to do with the current function. Instead, `this` is reserved for objects and always refers to the current object. There is no binding of `this`, applying `this`, or calling `this` in Dart. In other words, `this` has nothing to do with functions. We will discuss this again in [Classes and Objects](#), but only briefly because this is so dang simple in Dart!

3.4 What's Next

In many ways, this chapter is a work very much in progress. Dart lacks quite a lot of the power that is currently available to the JavaScript programmer: there is no reflection, and there is no arguments property available inside functions.

Even lacking these, we already see that Dart is extremely powerful in what it does allow us to do. We will return to this topic again in [Chapter 11, *Project: Varying Behavior*, on page 91](#).

Manipulating the DOM

We cannot write web applications without accessing and manipulating the DOM.¹ Sadly, Dart cannot do away entirely with established and often maddening aspects of the DOM API. Happily, Dart does provide a compatibility library that reestablishes some sanity when manipulating web pages.

4.1 dart:html

The library that we will use to interact and manipulate objects in a web page is `dart:html`. We will talk more about libraries in [Chapter 10, Libraries, on page 83](#). For now, just think of them like libraries from any other language (except JavaScript, of course)—a mechanism for encapsulating logically and physically separate functionality.

The `dart:html` library is not your grandmother's DOM API. The `dart:html` core library is Dart's take on what DOM programming should have been like from the beginning.

4.2 Finding Things

The primary entry points into DOM are `document.query()` and `document.queryAll()`. Both take a CSS selector as the argument. The former returns a single matching element; the latter returns a list of all matching elements. Here are some simple examples:

```
document.query('h1');  
// => First <h1> in the document  
  
document.query('#people-list');  
// => Element with id of 'people-list'
```

1. Document Object Model: http://en.wikipedia.org/wiki/Document_Object_Model

```
document.query('.active');
// => First element with 'active' class

document.queryAll('h2');
// => All <h2> elements
```

The `query()` and `queryAll()` methods are actually methods of the `Element` class. `Document`, like every other class representing a bit of the DOM, subclasses `Element`. In practice, this means we can limit queries to a specific element by first finding the element and then querying from it.

```
var list = document.query('ul#people-list');

var last_person = list.query(':last-child');

last_person.innerHTML
// => 'Bob'
```

Or we can chain it:

```
document.
  query('ul#people-list').
  query(':last-child').
  innerHTML;
// => 'Bob'
```

Important:

Although chaining `query()` methods might suggest a jQuery-like composability, Dart decidedly does not work with wrapped sets.

Consider trying to highlight the name of people in an unordered list. In jQuery, we might write something like this:

```
$('li', 'ul#people-list').
  attr('class', 'highlight');
```

In Dart, we have to manually iterate over each element.

```
document.
  query('ul#people-list').
  queryAll('li').
  each((li) {
    li.addClass('highlight');
  });
```

Although Dart makes working with the DOM easier than working with pure JavaScript, there are still some things that jQuery does a little better.

The bottom line with finding elements on a page in Dart is that `query()` and `queryAll()` do pretty much what you expect, making it easy to query the DOM.

4.3 Adding Things

Creating and adding new elements in an HTML document requires the combination of two Dart concepts: an `Element` and a `Node`. At their heart, node and element mean the same things in Dart that they do in typical DOM programming—nodes are the more general of the two. A node can represent an element, attributes of an element, and even text, whereas elements refer only to HTML tag objects.

To create a new element in Dart, we can use the handy `html` named constructor for `Element`.

```
var gallery = new Element.html('<div id="gallery">');
```

The `Element.html()` named constructor does not restrict us to creating a single element. We can create more complex HTML to be inserted into the page.

```
var gallery = new Element.html("""
<div id="gallery">
  <ul>
    <li>
    <li>
    <li>
    <!-- ... -->
  </ul>
</div>
""");
```

Dart's string interpolation is especially handy when building larger HTML fragments. The result can be almost template-like.

```
gallery(title, photographer) {
  return new Element.html("""
    <div id="gallery">
      <h2>${title}</h2>
      <ul>
        <li>
        <li>
        <li>
        <!-- ... -->
      </ul>
      <h3 class="footer">
        Photos by: ${photographer}
      </h3>
    </div>
    """);
}
```


To insert an element into the document, it is easiest to grab a `NodeList` to which the new element can be appended.

```
var gallery = new Element.html('<div id="gallery">');
document.
  query('#content').
  nodes.
  append(gallery);
```

When more sophisticated element insertion is needed, Dart supports the standard `insertAdjacentHTML()` method.² This is far more verbose than the jQuery `prepend()`, `append()`, `before()`, and `after()` Element methods. Even the somewhat shorter `insertAdjacentElement` suffers by comparison to the jQuery equivalent.

```
var gallery = new Element.html('<div id="gallery">');
document.
  query('#content').
  insertAdjacentElement('afterBegin', gallery);
```

The previous would insert the gallery element at the beginning of the content `<div>`'s nodes (equivalent to jQuery's `prepend()`).

Creating elements in Dart is quite nice. Appending them to the document or a list of nodes is also relatively easy. Dart still leaves much to be desired when we need to perform more sophisticated insertion of elements. Ideally, this will be improved in the near future.

4.4 Removing Things

Removing an element from the document is quite Darty.

```
document.
  query('#content').
  query('#gallery').
  remove();
```

The previous would find the element with the ID of "content"; then, inside that, it would find the "gallery" element and remove it from the page. If the page began like this:

```
<body>
<div id="content">
  <div id="gallery"/>
  <p class="instructions">...</p>
</div>
</body>
```

2. <https://developer.mozilla.org/en/DOM/Element.insertAdjacentHTML>

then the result of the previous `remove()` would be as follows:

```
<body>
<div id="content">
  <p class="instructions">...</p>
</div>
</body>
```

Not being required to walk up to an element's parent to remove the element, as we have to do in JavaScript, is a nice win.

4.5 Updating Elements

We have already discussed adding and removing elements from a parent element. Aside from those actions, the most common manipulation is adding and removing CSS classes on an element. The following would remove the "subdued" class from the `<blockquote>` tag and add the "highlighted" class to it:

```
document.
  query('blockquote').
    classes.
      remove('subdued');
```

```
document.
  query('blockquote').
    classes.
      add('highlighted');
```

Here, we again see the difference between Dart's set-based approach to classes and jQuery's chainable approach. In jQuery, removing and adding classes could be accomplished in a single statement with several chains. In Dart (for now) we are forced to use two separate statements.

In addition to manipulating classes, the `Element` class also allows for the familiar `innerHTML` change.

```
document.
  query('blockquote').
    innerHTML = 'Four score and <u>seven</u> years ago...';
```

Manipulating classes and updating `innerHTML` covers the most common cases of changing an element directly. If more is needed, then the `Element` class³ is the place to start looking.

Note: The `dart:html` core library has a corresponding `dart:htmlimpl` library. This "implementation" library contains the vast majority of concrete classes that are

3. <http://api.dartlang.org/html/Element.html>

used internally to represent actual objects in the DOM. The implementation library links to the underlying C code in the browser in order to perform the actual manipulation. For the most part, `dart:htmlimpl` can be ignored. Chances are, if you find yourself concerned with `dart:htmlimpl`, then something is quite wrong.

4.6 DOM Ready

There is no need for an on-DOM-ready handler in Dart. Dart makes the sane decision of deferring evaluation until the DOM is ready for processing.

Easy peasy!

4.7 What's Next

The Dart HTML library exposes a familiar yet powerfully different means for manipulating web pages. It is not a complete, high-level solution like jQuery, but it provides a much nicer foundation on which to build higher-level libraries.

Compiling to JavaScript

When Dart first came out, every major browser vendor, as well as the WebKit project, announced that they had absolutely no intention of embedding the Dart VM in their browsers. Many bristled at the very suggestion that a non-standard language be supported even obliquely. How another language was supposed to become a standard seemed a tricky question. Fortunately, Google had a plan.

In the grand tradition of CoffeeScript,¹ the Dart project includes a compiler capable of compiling Dart into JavaScript. The goal is that, even if Dart does not become an overnight standard, web developers tired of the quirks of JavaScript have a choice. We can now code in a modern language for the Web but still support the wide variety of browsers on the market.

The JavaScript generated by the Dart compiler includes shim code for the various Dart libraries. There is generated JavaScript that translates Dart DOM calls into JavaScript. There is generated JavaScript that supports Dart Ajax. For just about every feature of Dart, there is a corresponding chunk of JavaScript that gets included in the compiled output.

If that sounds large, well, it is. When first released, the compiler generated tens of thousands of lines of JavaScript!

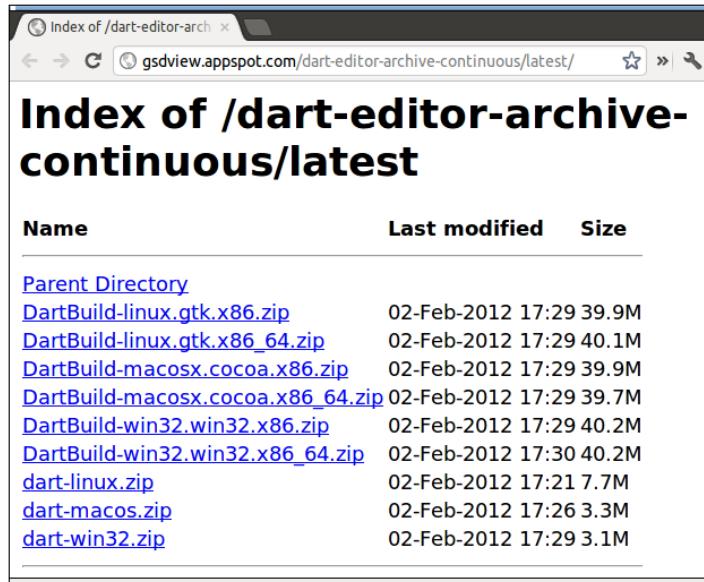
Of course, the compiler continues to get better. It still lacks compression/optimization, but already the compiler is producing JavaScript libraries in the range of thousands of lines of code instead of tens of thousands. Considering that Dart does a good chunk of the work of many JavaScript libraries like jQuery, this is already a good start.

And it is only going to get better.

1. <http://coffeescript.org/>

5.1 Compiling to JavaScript with dart2js

The tool provided to compile Dart down into JavaScript is dart2js. The dart2js compiler can be found among the Dart software development kit builds.² The SDK are the ones *without* “DartBuild” in the filename.



| Name | Last modified | Size |
|---|-------------------|-------|
| Parent Directory | | |
| DartBuild-linux.gtk.x86.zip | 02-Feb-2012 17:29 | 39.9M |
| DartBuild-linux.gtk.x86_64.zip | 02-Feb-2012 17:29 | 40.1M |
| DartBuild-macosx.cocoa.x86.zip | 02-Feb-2012 17:29 | 39.9M |
| DartBuild-macosx.cocoa.x86_64.zip | 02-Feb-2012 17:29 | 39.7M |
| DartBuild-win32.win32.x86.zip | 02-Feb-2012 17:29 | 40.2M |
| DartBuild-win32.win32.x86_64.zip | 02-Feb-2012 17:30 | 40.2M |
| dart-linux.zip | 02-Feb-2012 17:21 | 7.7M |
| dart-macos.zip | 02-Feb-2012 17:26 | 3.3M |
| dart-win32.zip | 02-Feb-2012 17:29 | 3.1M |

Ubuntu hipsters (truly the best hipsters) would use dart-linux.zip.

Once unzipped, we see that the SDK contains the entire Dart library (core, html, io, json).

```
.
+-- bin
+-- lib
|   +-- builtin
|   +-- core
|   +-- coreimpl
|   +-- dartdoc
|   +-- dom
|   +-- dart2js
|   +-- html
|   +-- io
|   +-- isolate
|   +-- json
|   +-- uri
|   +-- utf
+-- util
```

2. <http://gsdview.appspot.com/dart-editor-archive-continuous/latest/>

In addition to the core Dart libraries, the SDK also contains library code to support documentation (dartdoc) and compiling to JavaScript (dart2js).

Actually, using dart2js could not be more basic. Currently, there are no command-line switches that alter the behavior. We simply run the bin/dart2js script, giving it a single argument: the Dart code to be compiled.

```
$ dart2js main.dart
```

There is no output from the compiler indicating success, but we now have a nice little JavaScript version of our script.

```
$ ls -lh
-rw-rw-r-- 1 cstrom cstrom 462 2012-02-03 12:19 main.dart
-rw-rw-r-- 1 cstrom cstrom 7.0K 2012-02-03 12:19 main.dart.js
```

Well, maybe it's not "little."

If there are errors in the Dart code being compiled, dart2js does a very nice job of letting us know where the errors occur.

```
$ dart2js main.dart
main.dart:5:3: error: cannot resolve document
  document.query('#foo');
  ^^^^^^^
Error: Compilation failed.
```

One thing to bear in mind when compiling JavaScript is that dart2js works only at the application level, not the class level. Consider the situation in which we are converting our comic book collection application to follow a hip MVC pattern.

```
comics.dart
Collection.Comics.dart
HipsterModel.dart
Models.ComicBook.dart
Views.AddComic.dart
Views.AddComicForm.dart
Views.ComicsCollection.dart
```

There is no way to compile individual classes into usable JavaScript.

```
$ dart2js Models.ComicBook.dart
Models.ComicBook.dart:1:1: Could not find main
#library('Model class describing a comic book');
```

```
Error: Compilation failed.
```

If the script containing the main() entry point references the other libraries or if those libraries reference other libraries, then everything will be slurped into

the resulting JavaScript. The three libraries referenced in the following `#import()` statements will be pulled into the compiled JavaScript:

```
#import('ComicsCollection.dart');
#import('ComicsCollectionView.dart');
#import('AddComicView.dart');

main() { /* ... */ }
```

Similarly, the `ComicBook` model will also be included in the `dart2js`-generated JavaScript by virtue of being referenced in the collection class.

```
#library('Collection class to describe my comic book collection');

#import('Models.ComicBook.dart');

class ComicsCollection { /* ... */ }
```

At some point, it might be nice to write classes in Dart and compile them into usable JavaScript. For now, however, we are relegated to compiling entire applications, not pieces.

5.2 Maintaining Dart and JavaScript Side-by-Side

As Dart and `dart2js` evolve, the performance of the generated JavaScript will improve. At some point, compiled Dart code will rival and possibly surpass what the typical JavaScripter might write. But as fast as the compiled JavaScript gets, it will never be as fast as running Dart natively.

The question then becomes, how can we send Dart code to Dart-enabled browsers and send the compiled JavaScript to other browsers?

The answer is relatively simple: include a small JavaScript snippet that detects the absence of Dart and loads the corresponding JavaScript. As we saw in the previous section, if we compile a `main.dart` script, then `dart2js` will produce a corresponding `main.dart.js` JavaScript version.

The following JavaScript snippet will do the trick:

```
if (!navigator.webkitStartDart) loadJsEquivalentScripts();

function loadJsEquivalentScripts() {
  var scripts = document.getElementsByTagName('script');
  for (var i=0; i<scripts.length; i++) {
    loadJsEquivalent(scripts[i]);
  }
}
```

```
function loadJsEquivalent(script) {
  if (!script.hasAttribute('src')) return;
  if (!script.hasAttribute('type')) return;
  if (script.getAttribute('type') != 'application/dart') return;

  var js_script = document.createElement('script');
  js_script.setAttribute('src', script.getAttribute('src') + '.js');
  document.body.appendChild(js_script);
}
```

There is a similar script in Dart-core.³ In most cases, that script should be preferred over ours because it does other things (such as start the Dart engine).

The check for an available Dart engine is quite simple—if the browser knows how to start the Dart engine, then it is Dart-enabled.

```
if (navigator.webkitStartDart)
```

That may come in handy elsewhere in our Dart adventures.

The remainder of the JavaScript is fairly simple. The `loadJsEquivalentScripts()` function invokes `loadJsEquivalent()` for every `<script>` tag in the DOM. This method has a few guard clauses to ensure that a Dart script is in play. It then appends a new `.js <script>` to the DOM to trigger the equivalent JavaScript load.

To use that JavaScript detection script, we save it as `dart.js` and add it to the web page containing the Dart `<script>` tag.

```
<script src="/scripts/dart.js"></script>

<script src="/scripts/main.dart"
  type="application/dart"></script>
```

A Dart-enabled browser will evaluate and execute `main.dart` directly. Other browsers will ignore the unknown `"application/dart"` type and instead execute the code in `dart.js`, creating new `<script>` tags that source the `main.dart.js` file that we compiled with `dart2js`.

In the end, we have superfast code for browsers that support Dart. For both Dart and non-Dart browsers, we have elegant, structured, modern code. Even this early in Dart's evolution, we get the best of both worlds.

3. http://dart.googlecode.com/svn/branches/bleeding_edge/dart/client/dart.js

5.3 What's Next

The ability to compile Dart into JavaScript means that we do not have to wait for a tipping point of browser support before enjoying the power of Dart. Today, we can start writing web applications in Dart and expect that they will work for everyone. This is a good thing because, in the next chapters, we will be taking our simple Dart application to the next level and we wouldn't want to leave our nonhipster friends too far behind.

Part II

Effective Coding Techniques

With the basics of Dart out of the way, it is time to start exploring what makes Dart unique. We begin by converting the very simple application from Chapter 1 into a full-blown MVC client library. Happily, this is quite easy to do in Dart.

With the MVC library started, it is high time that we discuss some of Dart's most exciting features: the excellent object-oriented programming support and a very customizable events system.

Project: MVC in Dart

In this chapter, we will get our first real feel for what it means to write Dart code. Until now, our discussion has not strayed far from the familiar—or at least from what is similar to JavaScript.

We will take the very simple comic book collection application from [Chapter 1, Project: Your First Dart Application, on page 3](#), and we will convert it to an MVC design pattern. Since this will be client-based, it will not be Model-View-Controller. Rather, it will be Model-Collection-View, plus a Router, similar to Backbone.js.

We will start by implementing collections of objects in Dart and then describe the objects themselves. Once we have the foundation in place, we will take a look at views and templates.

This is another “project” chapter, so we will gloss over some Dart details to focus on writing code.

6.1 MVC in Dart

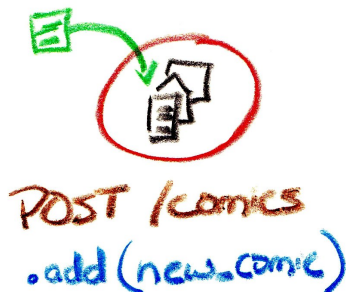
The foundation of our Hipster MVC library (of course that’s the name) will be collections of objects, not the objects themselves. The collection maps nicely onto REST-like web services, resulting in a clean API for adding, deleting, and updating records.

Harkening back to the first chapter, our comics collection can be retrieved via an HTTP GET of `/comics`. In Hipster MVC parlance, we will call that a `fetch()`.

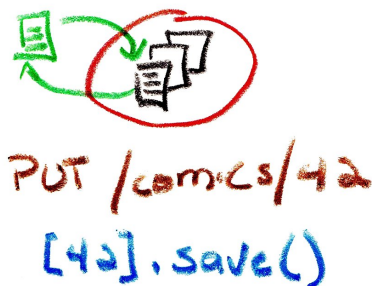


With REST-like resources, we can also refer to /comics as the URL root because it serves as the root for all operations on the collection of individual records.

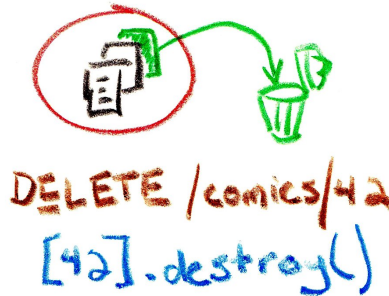
For instance, adding a new comic to the collection is an HTTP POST operation on /comics. And, in hipster-ese, that is an add().



To update a comic book with new information, we use HTTP's PUT. To identify the particular comic book being updated, we supply the ID in the subpath of the URL: PUT /comics/42. From Hipster MVC's perspective, we retrieve the record, update it, and save it with save().



Lastly, to remove a record from the collection, we use the destroy() method. This will result in an HTTP DELETE, again on the collection URL including the ID.



Let's get started writing that code.

6.2 Hipster Collections

Recall from [Project: Your First Dart Application](#) that our `main.dart` looks something like this:

`your_first_dart_app/comics_initial_main.dart`

```
#import('dart:html');
#import('dart:json');
main() {
  load_comics();
}
load_comics() {
  // Do stuff here
}
```

The `load_comics()` method retrieves the comic book collection from `/comics` and displays it on the web page.

In MVC, the collection object retrieves the records, and a view object displays the contents of the collection.

`mvc/main.dart`

```
#import('dart:html');
#import('dart:json');
main() {
  var my_comics_collection = new Comics()
    , comics_view = new ComicsView(
      el: '#comics-list',
      collection: my_comics_collection
    );
  my_comics_collection.fetch();
}
```

We will look at the view a little later. First, we define the collection class as follows:

mvc/collection_class.dart

```
class ComicsCollection implements Collection {
  // Hip Collection stuff goes here...
}
```

We declare our `ComicsCollection` class as implementing the `Collection` interface. Aside from the similarity in names, it makes sense to implement this interface because we will likely want our collection of comic books to behave like a `Collection`; we will want to have access to collection methods such as `filter()`, `forEach()`, `length`, and so on.

Note: When writing scripts or initial implementations of libraries, we can easily forgo typing information. When writing libraries that we hope others will use, it is a must. To be clear, it is possible to write reusable code without the type information, but it is tantamount to being a bad Dart citizen.

With the preliminaries out of the way, we describe the constructor for our `ComicsCollection` class. In Dart, constructor methods have the same name as the class.

mvc/collection_constructor.dart

```
class ComicsCollection implements Collection {
  CollectionEvents on;
  List models;
  // Constructor method
  ComicsCollection() {
    on = new CollectionEvents();
    models = [];
  }
}
```

We start by declaring two instance variables, `on` and `models`. Based on the type information, the `on` variable will hold an events list for the collection. We will explore this more in [Chapter 8, Events, on page 67](#). For now, it is enough to know that this object will be the nexus for subscribing to and generating custom events in our collections. The `models` property is a simple array to hold the list of objects in this collection.

The constructor itself takes no arguments but defines our two instance variables as blank states of `CollectionEvents` and `List` objects.

Next, we define some `Collection` methods.

mvc/collection_collection_methods.dart

```
class ComicsCollection implements Collection {
  // ...
  void forEach(fn) {
    models.forEach(fn);
  }
}
```

```

int get length() {
  return models.length;
}

operator [](id) {
  var ret;
  forEach((model) {
    if (model['id'] == id) ret = model;
  });
  return ret;
}
}

```

We cheat in our implementation of the `forEach()` method, delegating its implementation to our `models` instance variable. We do the same for the `length()` method, though we declare it as a getter. This allows us to access the method as `object.length` instead of `object.length()`. Reducing the number of parentheses in code is always a win.

Of the Collection-like methods declared, the most interesting is the `[]` operator. Dart supports numerous operators. This particular operator provides a means to supply an ID to look up a single object in the collection. That is, if we had a `comics` collection, the one with an ID of 42 is retrieved by `comics[42]`.

Now that we have our collection behaving like a `Collection`, let's make it behave like an Ajax-backed object. For discussion purposes, we will not go into complete CRUD but will focus on fetching the objects from the back-end data store, creating new objects in the data store and deleting them.

We already know from [Project: Your First Dart Application](#) how to fetch data over Ajax in Dart. In this case, when the data has loaded, we call the private `_handleOnLoad()` method.

`mvc/collection_fetch.dart`

```

class ComicsCollection implements Collection {
  // ...
  void fetch() {
    var req = new XMLHttpRequest();
    req.on.load.add((event) {
      var request = event.target,
          list = JSON.parse(request.responseText);
      _handleOnLoad(list);
    });
    // verb, resource, boolean async
    req.open('get', url, true);
    req.send();
  }
}

```

Instead of populating a UI element as we did in our first application, we need to behave in more frameworky fashion. That is, we build the internal collection and trigger events.

`mvc/collection_handle_on_load.dart`

```
class ComicsCollection implements Collection {
  // ...
  _handleOnLoad(list) {
    list.forEach((attrs) {
      var new_model = new ComicBook(attrs);
      new_model.collection = this;
      models.add(new_model);
    });

    on.load.dispatch(new CollectionEvent('load', this));
  }
}
```

For each set of model attributes, we create a new model object, set the model's collection property to our current collection, and add the model to the collection's models list. Once that is complete, we dispatch a load event for any object that might be interested in listening.

The model does not strictly need to know about the collection (in fact, it should not communicate directly with the collection). We assign it here so that the model can reuse the collection's URL for finding, creating, and updating back-end objects. The model will communicate with the collection via event broadcasting just as we have done here with our `on` property.

One of the events in the `CollectionEvents` object is the load event. Here we dispatch a `CollectionEvent` to anyone listening for notification that the collection has been loaded. As we will see in a bit, this is how we trigger an interested view object to draw itself.

Of course, we have not even introduced the `Model` base class yet, so let's get that out of the way next.

6.3 Hipster Models

Where the collection had a `models` property to store its data, the model will have the `attributes` property. Like the collection, the model will also need to expose an `on` attribute as the nexus for events generated by the model. Recall that the collection will pass the model a reference to itself, giving the model quick access to the collection's properties (for example, `url`).

Thus, we can begin defining the model class as follows:

mvc/model_constructor.dart

```
class ComicBook {
  Map attributes;
  ModelEvents on;
  HipsterCollection collection;

  ComicBook(this.attributes) {
    on = new ModelEvents();
  }
}
```

The declaration of the attributes, on, and collection properties in this class should be familiar now. Each results in a setter/getter for ComicBook instances (for example, comic_book.collection = my_comics and comic_book.attributes['title']). New here is the constructor that accepts an argument. Not only does it accept an argument, but writing the argument as this.foo has the same effect as if we had written the following:

```
class Foo
  var bar;

  Foo(bar) {
    this.bar = bar;
  }
}
```

Making use of this inside the argument list is a nice little Dart shortcut.

```
class Foo
  var bar;
  Foo(this.bar);
}
```

Anyway, since the model is Map-like, we use the [] operator to retrieve attribute values.

mvc/model_operator.dart

```
class ComicBook {
  // ...
  operator [](attr) => attributes[attr];
}
```

We will talk more about operators in [Chapter 7, Classes and Objects, on page 53](#), but the intent of that method is crystal clear. When we look up an attribute directly on the model (for example, comic_book['title']), the value in the attributes HashMap is returned. The hash-rocket shortcut for function bodies is extremely handy at times.

The URL for ComicBook is complicated only by the question of whether the model has an ID attribute. If it does have an ID, then we assume that the model has been previously saved in the back end. In this case, updates will be PUT against the resource root plus an ID (for example, /comics/42). Otherwise, this is a new model that will need to be POSTed to the resource root (for example, /comics). Recall also that if the model has a collection, then the URL root can come from the collection.

`mvc/model_url.dart`

```
class ComicBook {
  // ...
  get url() => isSaved() ?
    urlRoot : "$urlRoot/${attributes['id']}";

  get urlRoot() => 'comics';

  isSaved() => attributes['id'] == null;
}
```

The multiline ternary in `url()` and `urlRoot()` is a matter of taste. Written as shown previously, it almost read like dependencies in an old Makefile.

With that, we can now define the `save()` method. Saving a client-side model involves the following:

- An XMLHttpRequest object, over which the model data will be sent to the back-end datastore
- JSON functions to stringify the data before transport and to parse the response
- A listener for the XHR object's `on.load` (in other words, on success) event
- Replacing the model's attributes with data returned from the server
- Dispatching a “model-saved” event so that collections and views can update themselves appropriately
- Invoking an optional callback so that the object that called `save()` can respond appropriately

Whoa! There is a lot going on with a simple `save`, and the code reflects it.

`mvc/model_save.dart`

```
#import('dart:json');
class ComicBook {
  // ...
  save([callback]) {
    var req = new XMLHttpRequest()
    , json = JSON.stringify(attributes);
```

```

req.on.load.add((load_event) {
    var request = load_event.target;

    attributes = JSON.parse(request.responseText);

    var model_event = new ModelEvent('save', this);
    on.save.dispatch(model_event);
    if (callback != null) callback(model_event);
});

req.open('post', url, true);
req.setRequestHeader('Content-type', 'application/json');
req.send(json);
}
}

```

This should start to look quite familiar now. We create an XHR object, open it to POST to the model's url, set the HTTP header as JSON, and send the serialized model attributes. We also establish a request listener that, when it sees a successful load event, will update the model's attributes, dispatch its own model events, and invoke the callback, if supplied.

Lastly, we define a very familiar-looking `delete()` method.

```

mvc/model_delete.dart
class ComicBook {
    // ...
    delete([callback]) {
        var req = new XMLHttpRequest();

        req.on.load.add((event) {
            var request = load_event.target;

            var event = new ModelEvent('delete', this);
            on.delete.dispatch(event);
            if (callback != null) callback(event);
        });

        req.open('delete', "${url}", true);
        req.send();
    }
}

```

As with Backbone.js, the main means of communication between different parts of the MVC stack are events. Happily, Dart makes it quite easy to dispatch events: `on.delete.dispatch(event)`. Listening for these events is just as easy, as we will see in our views.

6.4 Hipster Views

Of the three parts to our minimalistic MVC library, the view is easily the most lightweight. As mentioned earlier, by virtue of the Precipitation Pattern, the view sits atop the MVC stack. As such, it is allowed to communicate directly with either a model or a collection (both at the same time would be frowned upon). The view can update a model based on user input, and the view needs to know the values of a model's attributes so that it can display them. Thus, the view needs to expose both a collection property and a model property.

`mvc/view_properties.dart`

```
class ComicsView {
  var collection, model;
  Element el;
}
```

Each of these properties is optional in a view—it will be a collection view subclass's responsibility to know that it is a collection view and, as such, that it needs to access the collection property. Dart has a mechanism for handling optional parameters in functions and constructors. We use it here to declare `el`, `model`, and `collection` as optional:

`mvc/view_constructor.dart`

```
class ComicsView {
  // ...
  ComicsView([this.el, this.model, this.collection]);
}
```

For one line of code, there is a lot of information being conveyed here. Let's look at the `el` property first. An already existing DOM element can be injected via an `el` named parameter: `new ComicsView(el: '#comics-list')`. In Dart, optional parameters that are declared inside square brackets are named parameters. We name parameters by following the name with a colon and the value that we want to assign.

A very cool feature of optional parameters is that we do not need to explicitly assign them if they correspond to instance variables. To indicate that the optional variables correspond to instance variables of the same name, we declare them in the constructor's parameter list as `this.model` and `this.collection`. To inject a collection into a view object, we can use the collection named parameter: `new ComicsView(collection: comics_collection)`.

At this point, we have a view object that does not yet fulfill its purpose: rendering to the UI. To actually render the view, a `render()` method can be defined such that it assigns the innerHTML of the view's `Element el` to the result of applying

the template to the collection. In this case, the template does no more than iterate over the entire collection, applying a single comic book template to each model.

`mvc/view_collection_render.dart`

```
class ComicsView {
  // ...
  render() {
    el.innerHTML = template(collection);
  }

  template(list) {
    if (list.length == 0) return '';

    var html = '';
    list.forEach((comic) {
      html += _singleComicBookTemplate(comic);
    });
    return html;
  }

  _singleComicBookTemplate(comic) {
    return ""
      <li id="${comic['id']}">
        ${comic['title']}
        (${comic['author']})
        <a href="#" class="delete">[delete]</a>
      </li>"";
  }
}
```

The `_singleComicBookTemplate()` private method is an interesting little method. It uses Dart's multiline strings and interpolation to produce a close facsimile of a traditional template.

We will not explicitly invoke the `render()` method. Instead, we subscribe the view to the collection's events. Whenever the collection is loaded or updated, we will want to refresh the view. To accomplish this, the constructor needs a body that will subscribe to events.

`mvc/view_constructor_subscribe.dart`

```
class ComicsViews {
  // ...
  ComicsView([this.el, this.model, this.collection]) {
    _subscribeEvents();
  }
}
```

We care about only two events so far: add and load. When the collection is first loaded or when a new record is added to the collection, we declare that we want the view to rerender itself.

`mvc/view_collection_subscribe_events.dart`

```
class ComicsView {
  // ...
  _subscribeEvents() {
    if (collection == null) return;

    collection.on.load.add((event) { render(); });
    collection.on.add.add((event) { render(); });
  }
}
```

The API for adding event listeners is simple and expressive. More importantly, there is no need to worry about this. Even inside the anonymous functions invoked by the event listeners, methods continue to be invoked on the containing class. The elimination of `apply()`, `call()`, and `bind()` complexity is a definite benefit in this case.

Surprisingly, that is all that is needed to render the template. Our `main()` entry point creates a collection object, gives it to the view, and then performs a `fetch()`.

`mvc/main.dart`

```
#import('dart:html');
#import('dart:json');
main() {
  var my_comics_collection = new Comics()
    , comics_view = new ComicsView(
      el: '#comics-list',
      collection: my_comics_collection
    );
  my_comics_collection.fetch();
}
```

When `fetch()` completes, it broadcasts a load event at which point the view renders itself.

At this point, we have entirely replicated our original application in an MVC approach. Unlike the original implementation, this approach is aware of how to add and remove items from the back end. Let's add the ability to remove a comic book from the collection.

6.5 Working with MVC to Delete

To enable the user to delete a record from our MVC application, we need to attach event handlers to the UI. This can be accomplished via another call during initialization—this time to an `_attachUiHandlers()` method.

```
mvc/view_collection_attach_ui_events.dart
class ComicsViews {
  // ...
  ComicsView([this.el, this.model, this.collection]) {
    _subscribeEvents();
    _attachUiHandlers();
  }
  // ...
  _attachUiHandlers() {
    attach_handler(el, 'click .delete', delete);
  }
  delete(event) {
    var id = event.target.parent.id;
    collection[id].delete(callback:(_) {
      event.target.parent.remove();
    });
  }
}
```

Here, we delegate click events on elements with `class="delete"` to the view's `delete()` method. The ID of the current object in the collection is stored in the parent element's ID attribute (we could also have used an HTML5 data-attribute). Using the ID, we find the model in the collection so that we can invoke the model's `delete()` method. We supply a callback so that when the model is successfully removed from the back-end data store, we can remove the element from the web page as well.

This is when custom delegated events come in handy. Since the collection may not have been initialized when the class is evaluated and since new elements will likely be added to the view, it is an absolute necessity to be able to define event handlers that work now and while the page gets updated.

Delegated events are not (yet) built into Dart, but we can add them.

```
mvc/view_attach_handler.dart
class ComicsView {
  // ...

  attachHandler(parent, event_selector, callback) {
    var index = event_selector.indexOf(' ')
    , event_type = event_selector.substring(0,index)
    , selector = event_selector.substring(index+1);
```

```

parent.on[event_type].add((event) {
  var found = false;
  parent.queryAll(selector).forEach((el) {
    if (el == event.target) found = true;
  });
  if (!found) return;

  callback(event);

  event.preventDefault();
});
}
}

```

When we say `attach_handler(el, 'click .delete', delete)`, we are attaching a handler to the `el` DOM element for specific events. The `event_type` in this case is `click`, and the selector is the `.delete` CSS class. We add a click event listener on the parent element, `el`. Every time a click is registered on that element, `attachHandler()` queries all of the children looking for one matching `.delete`. If found, the supplied callback is then invoked.

That Dart does not support delegated events yet is, ideally, another early omission that will be rectified as the language evolves. In the meantime, our simple solution will suffice. We can now delete comic books from the UI.

6.6 What's Next

Phew! We really put Dart through its paces in this chapter. We took our very first Dart application and converted it to a very functional MVC library. We also caught a few glimpses of both object-oriented and event-based programming. At times we glossed over the details of Dart's support for those two programming paradigms, so we will take the next two chapters to explore them in more detail.

When we pick up our project again in [Chapter 9, Project: Extracting Libraries, on page 75](#), we will see one of Dart's absolute coolest early features: real libraries.

Classes and Objects

[Chapter 6, *Project: MVC in Dart*, on page 39](#) made extensive use of classes and objects to build up an MVC library. From this, we can take two lessons: first, it is hard to do significant work in Dart without some object-oriented programming, and second, it is pretty easy to do object-oriented programming in Dart. In this chapter, we will formalize how Dart treats its classes and objects.

7.1 Class Is a First-Order Concept

Dart's classical approach to object-oriented programming is a significant, and welcome, departure from JavaScript's prototype-based approach. Prototype-based languages certainly offer some benefits, but ease of approach is not one of them.

As we have seen, Dart classes are introduced with the `class` keyword.

```
class ComicsCollection {  
    // describe class operations here...  
}
```

That is all we need in order to define a class in Dart—no fancy constructor functions, no heavy libraries to provide classical classes, just `class` followed by a class name.

Note: Although a class is a first-order concept in Dart, it is not a first-order *object*. As we saw in [Project: MVC in Dart](#), it is not possible to pass a class name as a variable as we might do in other languages.

7.2 Instance Variables

Instance variables are nothing more than variables declared inside a class.

```
class ComicsView {
  ComicsCollection collection;
  ComicsModel model;
  Element el;
  // ...
}
```

In this case, we have declared three instance variables of different types. We could have declared all three as having a variable type (`var collection, model, el;`), but we are being good library maintainers by being explicit about the types.

Instance variables are accessible throughout the class with just their name—there is no need to prepend `this.` to a variable unless it would help to disambiguate an instance variable from a local variable.

By default, instance variables are public, which means they are accessible by subclasses *and* outside the class. For any public instance variable, Dart automatically creates an external getter and setter with the same name as the instance variable. For example, to access the comic book view’s collection, we can do this:

```
comics_view.collection;
// => instance of ComicsCollection
```

To switch the view to a new collection, we can do this:

```
comics_view.collection = new_collection;
```

Public instance variables are a nice convenience but should be used with caution if access control is needed.

Private Instance Variables

In some cases, public instance variables are a scary proposition. If a library does not want to expose an instance variable directly to its consuming context, then it can declare private variables. Private variables in Dart are simply variables that start with an underscore (for example, `_models`). If, for example, we did not want to allow the collection to be changed, we could declare the collection as a private instance variable but still expose a public “getter.”

```
class ComicsView {
  // Private because it starts with underscore
  ComicsCollection _collection;
  ComicsCollection get collection() {
    // possibly restrict access here...
    return _collection;
  }
}
```

Important: Private instance variables are available *only* to the enclosing class. A superclass does not have access to its concrete class's private variables, and vice versa. We will run headlong into this restriction in [Chapter 11, Project: Varying Behavior, on page 91](#). Another way of thinking of this restriction is that if you need to override an instance variable, it cannot be private.

7.3 Methods

Methods are simply named functions inside a class. The following is a method that renders the current view by assigning the `el` instance variable's `innerHTML` to the result of a template.

```
class ComicsView {
  // ...
  render() {
    el.innerHTML = template(collection);
  }
}
```

Inside the class, methods may be invoked by calling the method directly—prepending the method with `this.` is not required like it is in JavaScript (though it would still work). In the previous example, `template()` is an instance method that is invoked with the view's `collection` property.

In Dart, it is generally considered good practice to prepend the return type of a method or prepend `void` if the method does not return anything.

```
class ComicsView {
  // ...
  void render() {
    el.innerHTML = template(collection);
  }
}
```

In addition to “normal” methods, Dart also supports specialized setter and getter methods as well as operator methods.

Getters and Setters

Getter methods are those that take no arguments and are declared with the keyword `get` after the type and before the name of the method.

```
class ComicsCollection extends Collection {
  String get url() => '/comics';
}
```

Getters get their name from how they are used, which greatly resembles getting an object's property in other languages.

```
// No parens required!
comics_collection.url;
// => '/comics'
```

Dart also supports setters, which are functions that assign new values. These are declared with the `set` keyword and are methods that accept a single parameter (the newly assigned value).

```
class ComicsCollection extends Collection {
  String _url;
  void set url(new_url) {
    _url = new_url;
  }
  String get url() => _url;
}
```

Setters are of interest primarily because they override the assignment operator. To set the new URL in the Comics class, we would not pass it as the argument to the `url()` method. Rather, we assign it.

```
comics_collection.url = shiny_new_url;
```

Dart recognizes assignment as a special setter operation and looks in the class definition for the `set` keyword to decide how to proceed.

Setters and getters beat the pants off of languages that force us to choose from any number of poor conventions to indicate intent.

Operators

In fact, there are a number of operator-like methods that can be described by a Dart class. The remaining operators are declared with the same keyword: `operator`.

We saw `operator` in the `ComicsModel` class as a way to access attributes of the model.

```
class ComicsModel {
  // ...
  operator [](attr) => attributes[attr];
}
```

With that, we can then look up the title of a `Comic` object with the following:

```
comic['title'] // => "V for Vendetta"
```

The square bracket lookup is by far the most common operator in Dart, but a myriad are supported: `==`, `<<`, `>`, `<=`, `>=`, `-`, `+`, `/`, `~/`, `*`, `%`, `|`, `^`, `&`, `<<<`, `>>>`, `[]=`, `[]`, `~`, and `!`.

There is also a special call keyword that lets us describe what should happen when an object is applied. For instance, if we want calling a model (*e.g.* `comic()`) to be an alias for saving it, we could declare it as follows:

```
class ComicsModel {
  operator call() {
    this.save();
  }
}
```

Then, saving could be accomplished with the following:

```
comic_book();
```

Metaprogramming with `noSuchMethod`

In its early days, Dart provides for limited metaprogramming facilities. One vehicle for dynamically changing behavior at runtime is the special method `noSuchMethod()`. When Dart attempts to locate a method being called, it first looks in the current class for the explicit definition. If the method is not located, the superclass and all ancestor classes are checked. Failing that, Dart then invokes `noSuchMethod()`—if it has been declared—in the current class.

When invoked, `noSuchMethod()` receives the name of the method being invoked and the list of arguments being passed.

```
class ComicsModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }
    // Do save operations here...
  }
}
```

We will look into `noSuchMethod()` in more detail in [Project: Varying Behavior](#).

Note: Dart will likely add more dynamism as it evolves, but it is not a priority for two reasons.

1. It adversely affects code completion.
2. It is a common source of bugs that the compiler cannot identify.

For those of us who are not fans of code completion, #1 is not a strong argument. Ruby and JavaScript programmers might contend with #2—the idea that dynamic language features are a common source of bugs. Even so, they certainly prevent the compiler from catching potential issues.

Regardless, Dart is not opposed to becoming more dynamic in the future; it is just not an immediate priority.

7.4 Static Methods and Variables (aka Class Methods and Variables)

Dart includes the concept of class variables and methods, though it takes a dim view of them. It regards them as a necessary evil, which, of course, they are. These are introduced with the static keyword.

```
class Planet {
  static List rocky_planets = const [
    'Mercury', 'Venus', 'Earth', 'Mars'
  ];

  static List gas_giants = const [
    'Jupiter', 'Saturn', 'Uranus', 'Neptune'
  ];

  static List known() {
    var all = [];
    all.addAll(rocky_planets);
    all.addAll(gas_giants);
    return all;
  }
}
```

Invoking a static method is just like invoking an instance method, except the class itself is the receiver.

```
Planet.known()
// => ['Mercury', 'Venus', 'Earth', 'Mars',
//      'Jupiter', 'Saturn', 'Uranus', 'Neptune' ]
```

Interestingly, instance methods can treat static methods as if they are other instance methods.

```
class Planet {
  // ...
  static List known() { /* ... */ }

  String name;
  Planet(this.name);

  bool isRealPlanet() {
    return known().some((p) => p == this.name);
  }
}
```

In the previous code, the instance method `isRealPlanet()` invokes the static method `known()` just like it would any instance method. In this way, we can find that Neptune is a real planet, but Pluto is not.

```

var pluto = new Planet('Pluto');
var neptune = new Planet('Neptune');

neptune.isRealPlanet()
// => true

pluto.isRealPlanet();
// => false

```

Warning: Because Dart treats static methods as instance methods in this fashion, it is illegal to have an instance method with the same name as a class method.

7.5 Interfaces

Interfaces in Dart are used to describe functionality shared in common by concrete classes. They describe a set of methods that classes should implement. They also provide a means to limit the type of arguments that are supported in function and method calls.

Our ComicsCollection class implements the built-in Collection interface.

```

class ComicsCollection implements Collection {
  void forEach(fn) {
    models.forEach(fn);
  }

  int get length() {
    return models.length;
  }
}

```

What this tells other Dart classes is that there is a reasonable expectation that this class supports Collection methods like `forEach` and `length`.

Dynamic language adherents fancy duck typing, which is the equivalent of asking “Who cares what the type is as long as the object supports `forEach`?” In fact, Dart will let you get away with this kind of behavior if you like. That said, you are a better Dart citizen if you use an interface to declare *why* you support particular methods.

If you need to support multiple interfaces, simply separate them with commas in the class declaration.

```

class ComicsCollection implements Collection, Hashable, EventTarget {
  // Collection methods
  void forEach(fn) { /* ... */ }
  int get length() { /* ... */ }
}

```

```
// Hashable method
static String hash() { /* ... */ }

// EventTarget
Events get on() => _on;
}
```

To be clear, *none* of this is necessary in Dart, but judicious use of interfaces goes a long way toward improving the readability of code.

7.6 Subclasses

In Dart, we say that a subclass *extends* its superclass with new functionality. As we will see in [Chapter 9, Project: Extracting Libraries, on page 75](#), most of a collection’s functionality can be factored out into a `HipsterCollection` superclass. The `Comics` subclass then needs to extend `HipsterCollection` with only a few methods.

```
class Comics extends HipsterCollection {
  get url() => '/comics';
  modelMaker(attrs) => new ComicBook(attrs);
}
```

The `extends` keyword has the obvious benefit of reading nicely, which increases a codebase’s overall maintainability.

Note: Currently, the only way for Dart classes to implement multiple behaviors is by implementing multiple interfaces. Classes cannot subclass more than one superclass. Mixins (or *traits*) are still a work in progress.

Abstract Methods

In the previous code example, both `url` and `modelMaker` are abstract methods in the base class.

```
class HipsterCollection {
  abstract HipsterModel modelMaker(attrs);
  abstract String get url();
}
```

This indicates that `HipsterCollection` is an abstract class (that it will not work without a subclass) and is one that ideally overrides these methods (there is no abstract modifier for classes). If a subclass does not implement these methods, the code will not throw a compile-time error. However, a not-implemented exception is sure to follow.

7.7 Constructors

Dart gets a surprising amount of mileage out of its constructors. It does so through a combination of two types of constructors: generative and factory. The difference between the two has to do with how they create new objects. Generative constructors take care of blessing new objects for us, leaving us the task of initializing the internal state. In a factory constructor, we are responsible for building and returning new objects ourselves. As we will see, there is power in both the simplicity of generative constructors and the flexibility of factory constructors.

Generative constructors are the more common of the two, so we will talk about them first.

Simple Generative Constructors

Borrowing from [Project: MVC in Dart](#), the simplest form of a constructor looks a lot like a method with the same name as the class.

```
class ComicsCollection {
  CollectionEvents on;
  List<ComicsModel> models;

  // Our constructor
  ComicsCollection() {
    on = new CollectionEvents();
    models = [];
  }
}
```

The constructor for this class accepts no arguments and assigns two instance variables to default values. There is no return from a generative constructor—we affect only the internal state of an object.

Named Constructors

JavaScript is able to accomplish a lot with the arguments object/array. Dart does not have any support for this concept. It makes up for this with optional parameters, which we met in [Chapter 3, Functional Programming in Dart, on page 19](#), and named constructors. *Named constructors* are a mechanism for creating a specialized constructor. For instance, if we wanted to be able to create a `ComicsCollection` from a list of attributes, we could declare a `ComicsCollection.fromCollection` constructor.

```
class ComicsCollection {
  ComicsCollection() { /* ... */ }
```

```

ComicsCollection.fromCollection(collection) {
  on = new CollectionEvents();
  models = [];
  collection.forEach((attr) {
    var model = modelMaker(attr);
    models << model;
  })
}
// ...
}

```

This lets us instantiate a collection object thusly:

```

var comics_collection = new ComicsCollection.fromCollection([
  {'id': 1, 'title': 'V for Vendetta', /* ... */ },
  {'id': 2, 'title': 'Superman', /* ... */ },
  {'id': 3, 'title': 'Sandman', /* ... */ }
]);

```

Just like “normal” generative constructors, named constructors begin with the name of the class. The named constructor is denoted with a dot and then the name (for example, `.fromCollection`). And, just as with other generative constructors, named constructors do not return anything; they merely change the internal state of an object.

Classes can have any number of named constructors in addition to the normal constructor. This allows us to specialize object instantiation through a series of well-named constructors that do one thing. This effectively eliminates the complex conditionals that can plague object initialization in JavaScript.

Named constructors are a *huge* win for readable, maintainable code.

Redirecting Constructors

Once we start making effective use of Dart’s multiple constructors, we quickly get into a situation in which we are repeating logic. For instance, two different constructors for the model base class need to establish the `on` property for listening and broadcasting events.

```

class ComicsModel {
  Map attributes;
  ModelEvents on;
  ComicsModel(this.attributes, [this.collection]) {
    on = new ModelEvents();
  }
  ComicsModel.fromMap(this.attributes) {
    on = new ModelEvents();
  }
}

```

Note: We cannot simply assign on to new ModelEvents() when it is declared because it is not a compile-time constant. Strong typing giveth and strong typing taketh away. This is one of the cases in which it taketh away.

To avoid repeating ourselves, we use redirection constructors.

```
class ComicsModel {
  Map attributes;
  ModelEvents on;

  ComicsModel(this.attributes, [this.collection]) {
    on = new ModelEvents();
  }

  // Redirect the fromMap constructor to the
  // all-purpose constructor
  ComicsModel.fromMap(attributes): this(attributes);
}
```

Now, the on attribute is declared in only one place—the default new ComicsModel() constructor. That is a nice little maintainability win.

Redirection is introduced with the colon. The target of the redirection follows the colon; in this case, we specify the default constructor with this(). Redirection can point to other named constructors: this.withTitle(title). It can also point to the superclass constructor or named superclass constructors. For instance, the ComicBook model might need to define constructors for more prolific authors.

```
class ComicBook extends ComicsModel {
  ComicBook(): super();
  ComicBook.byNeilGaiman(): super({author: 'Neil Gaiman'});
}
```

Important: The previous example demonstrates implied constructors. If no constructors are defined, Dart adopts an implicit ComicBook(): super() redirection constructor. As soon as we define a constructor in the subclass—the .byNeilGaiman() constructor in this case—then there is no implied constructor, and we are forced to make the implicit constructor explicit.

Constructor Arguments

We have already seen an example of supplying arguments to named constructors. Regular constructors are no different.

```
class ComicsModel {
  Map attributes;
  ModelEvents on;

  ComicsModel(attributes) {
    this.attributes = attributes;
```

```

        on = new ModelEvents();
    }
    // ...
}

```

Dart provides a nice convention for assigning instance variables. Instead of assigning `this.attributes` in the constructor block as shown earlier, we can declare the parameter as `this.attributes`.

```

class ComicsModel {
    Map attributes;
    ModelEvents on;
    ComicsModel(this.attributes) {
        on = new ModelEvents();
    }
    // ...
}

```

This goes a long way toward clearing up intent. Rather than muddying up the assignment of the `attributes` instance variable alongside other constructor initialization and assignment, the intent is made quite clear in the parameter list. The body of the constructor can then concern itself solely with doing what it needs to in order to create an instance of the class.

This convention of declaring instance variables in the parameters list even works with optional parameters.

```

class ComicsView {
    ComicsCollection collection;
    ComicsModel model;
    Element el;

    ComicsView([el, this.model, this.collection]) {
        if (el != null) {
            this.el = (el is Element) ? el : document.query(el);
        }

        this.post_initialize();
    }
    // ...
}

```

In this case, the constructor has a bit of “real” work to do (querying the document for an element if one is not supplied). By declaring `this.model` and `this.collection` as optional parameters, the intent of assigning them is clear—without adding clutter to the constructor body.

Optional assignment parameters are passed with the name of the instance variable being assigned.

```
var comics_view = new Views.Comics(
  el: '#comics-list',
  collection: my_comics_collection
);
```

Like named constructors, Dart's parameter assignment goes out of its way to help keep your code clean.

Factory Constructors

Dart defines a special class of constructors for returning specialized object instances. The constructors that we have seen so far all manipulate the internal state of the newly created object but leave the blessing of the object and the return value to Dart.

```
class ComicsCollection {
  ComicsCollection() {
    on = new CollectionEvents();
    models = [];
  }
  // ...
}
```

If we instantiate an object via `new ComicsCollection()`, then we are returned an object of `ComicsCollection` with `on` and `models` instance variables started in their pristine states. This covers 90 percent of object-oriented programming, but there are times when we might want more.

For instance, what if we do not want to create a new object? What if our class should return a cached copy of a previously assembled object? What if we need our class to always return the same instance? What if we need the constructor to return a different object entirely? Dart defines factory constructors to answer all of those questions. The two syntactic differences between factory and normal constructors are the factory keyword and the return value of the constructor. Consider the factory constructor for the Highlander class.

```
class Highlander {
  static Highlander the_one;
  String name;
  factory Highlander(name) {
    if (the_one == null) {
      the_one = new Highlander._internal(name);
    }
    return the_one;
  }
  // private, named constructor
  Highlander._internal(this.name);
}
```

The Highlander's factory constructor checks to see whether the class variable `the_one` has already been defined. If not, it assigns it to a new instance via a private, named constructor and returns `the_one`. If `the_one` has already been defined, then no new instance is created, and the previously defined `the_one` is returned.

Thus, we have created a singleton class.

```
var highlander = new Highlander('Connor Macleod');
var another = new Highlander('Kurgan');

highlander.name
// => 'Connor Macleod'

// Nice try Kurgan...
another.name
// => 'Connor Macleod'
```

The utility of factory constructors is not limited to returning cached copies of the current class. They can return any object.

```
class PrettyName {
  factory PrettyName(name) {
    return "Pretty $name";
  }
}
```

Instances of this simple class would be *strings* with “Pretty” prefixed to the name.

```
new PrettyName("Bob");
// => "Pretty Bob"
```

Used wisely, there is much power in these beasts.

7.8 What's Next

Dart provides some very nice object-oriented programming features. Most of the emphasis from the language seems to be geared toward making the resulting code cleaner and the intent clearer. Although it supports the `this` keyword, representing the current object, its use is *far* less prevalent than in JavaScript, and the rules surrounding it far are less arcane. Effective use of generative, factory, and redirecting constructors goes a long way toward making our Dart code as clean as possible.

We will revisit classes in [Project: Varying Behavior](#). More specifically, there are implications for some of what we discussed that are better brought up in the context of real-world use.

Events

Regardless of the language, browser events are captured and received and bubble the same way. So, it makes sense that events in Dart behave in a fashion similar to JavaScript...with a few Darty twists.

8.1 Plain-Old Events

Consider, for instance, a click handler that colors the border of the clicked element a brilliant orange.

```
var el = document.query('#clicky-box');
el.on.click.add((event) {
  el.style.border = "5px solid orange";
});
```

Here, we are adding an anonymous function to the list of callbacks invoked when a click event occurs. This is slightly more compact than the equivalent JavaScript: `el.addEventListener('click', callback_fn)`. But there are implications of Dart's syntax, especially since Dart is strongly typed.

Working through the chain that adds an event listener, the first thing that we come across is the `on` property of an `Element`. The `on` attribute is an `ElementEvents` object, which exposes a number of different getters corresponding to each event supported.

The `ElementEvents` class used to define `on` is a subclass of `Events`, so we can expect patterns similar to `thing_with_events.on.event_name` when working with other evented classes. The `XMLHttpRequestEvents` class, for example, exposes event handlers like `error`, `load`, and `progress` on its `on` property. In fact, we saw an example of this in [Chapter 1, Project: Your First Dart Application, on page 3](#). The `on.load` Ajax handler looks like this:

```
req.on.load.add((event) {
  var list = JSON.parse(req.responseText);
  container.innerHTML = graphic_novels_template(list);
});
```

Ajax event handling and element event handling look the same because, you guessed it, both `Element` and `XMLHttpRequest` implement the same interface: `EventTarget`. The `EventTarget` interface demands nothing more of its implementers than an `on` getter that itself implements `Events`. It is this kind of organization that puts the “structured” in Dart’s structured code for the modern Web.

Continuing to work through the `el.on.click.add()` chain that adds a listener, we are up to `click`. The `click` getter, like any other property of `on`, returns an `EventListenerList`. This list exposes three very important methods: `add()`, `remove()`, and `dispatch()`. The first two obviously add and remove listeners in the `EventListenerList`. The `dispatch()` method is how we can manually trigger events.

If we wanted to trigger a click event, we would do something like this:

```
el.on.click.dispatch(new Event("My Event"));
```

Note: It is generally good practice to supply an `Element` as the event target when dispatching element events.

8.2 Custom Event Systems

A number of different event systems are built into Dart. So far in this chapter, we have explored the `ElementEvents` system. Not including subinterfaces, there are at least seven event systems:

- `XMLHttpRequestEvents`
- `WindowEvents`
- `DOMApplicationCacheEvents`
- `AbstractWorkerEvents`
- `EventSourceEvents`
- `XMLHttpRequestUploadEvents`
- `ElementEvents`

To build our own events system, we need to follow the same convention of exposing an `on` property. As with the built-in `EventTarget` objects, this `on` property will expose a number of custom event types to which event listeners can be added and removed. That is, it needs to expose a number of different `EventListenerList` getters.

Revisiting the ComicsCollection class, we have already seen that the on property is an instance of CollectionEvents.

```
class ComicsCollection implements Collection {
  CollectionEvents on;
  ComicsCollection() {
    on = new CollectionEvents();
  }

  // ...
}
```

This CollectionEvents is of our own design. It implements the built-in Events interface and exposes load and insert as event listeners.

```
class CollectionEvents implements Events {
  CollectionEventList load_listeners, insert_listeners;
  CollectionEvents() {
    load_listeners = new CollectionEventList();
    insert_listeners = new CollectionEventList();
  }
  CollectionEventList get load() => load_listeners;
  CollectionEventList get insert() => insert_listeners;
}
```

Whenever a new instance of CollectionEvents is created, this constructor builds two instances of CollectionEventList—one to hold the list of on.load event listeners and the other to hold the list of listeners waiting for a new item to be inserted into the collection.

As for the CollectionEventList class that describes on.load and on.insert, we need to implement the EventListenerList interface. This is a relatively straightforward class whose primary purpose is to maintain a list of callbacks to be fired when certain events occur. As mentioned earlier in the chapter, the EventListenerList interface mandates that we define add() and remove() methods to add and remove callbacks to the internal list of listeners. It also needs to define a dispatch() method that can invoke all listeners when an event is generated.

```
class CollectionEventList implements EventListenerList {
  List listeners;

  CollectionEventList() {
    listeners = [];
  }

  CollectionEventList add(fn) {
    listeners.add(fn);
    return this;
  }
}
```

```

    bool dispatch(CollectionEvent event) {
        listeners.forEach((fn) {fn(event);});
        return true;
    }
}

```

Of particular note in the definition of `CollectionEventList` is that the `add()` method returns the current instance of itself. This allows us to add multiple callbacks at one time.

```

hipster_collection.
  on.
  insert.
  add((event) { /* listener #1 */ }).
  add((event) { /* listener #2 */ }).
  add((event) { /* listener #3 */ });

```

Note: This does *not* allow us to add different types of events in a single statement. In the previous example, `add()` returns the instance of `CollectionEventList` that is dedicated to handling insert events. To define a series of load callbacks, we would need a separate `hipster_collection.on.load.add()` statement.

With that, we can finally take a look at the actual event being generated. The only thing that really needs to be defined in a custom event is the type getter, which will usually correspond to the names of the event listener list (for example, `load`, `insert`). To serve the needs of `ComicsCollection`, we can also collect the collection itself and, optionally, the model (which might be handy when inserting or removing).

```

class CollectionEvent implements Event {
    String _type;
    ComicsCollection collection;
    ComicsModel _model;

    CollectionEvent(this._type, this.collection, [model]) {
        _model = model;
    }

    String get type() => _type;

    ComicsModel get model() => _model;
}

```

To be a good Dart citizen, this class should also expose a target method.

```

class CollectionEvent implements Event {
    // ...
    EventTarget get target() => collection;
}

```

This, in turn, suggests that our ComicsCollection class implements the EventTarget interface.

```
class ComicsCollection implements Collection, EventTarget {
  // ...
}
```

We have already seen how this event system is used. The ComicsCollection base class dispatches its events when new models are added or when the load from the back end is complete.

```
class ComicsCollection implements Collection {
  // ...
  add(model) {
    models.add(model);
    on.add.
      dispatch(new CollectionEvent('add', this, model:model));
  }
  _handleOnLoad(list) {
    // add operations here
    on.load.dispatch(new CollectionEvent('load', this));
  }
}
```

To listen to these events, the Comics view collection renders itself when the collection is loaded from the back-end data store or whenever a new item is added to the collection.

```
class ComicsView {
  // called by the constructor
  _subscribeEvents() {
    if (collection == null) return;
    collection.on.load.add((event) { render(); });
    collection.on.add.add((event) { render(); });
  }
}
```

The main benefit of evented approaches like this is an elegant separation of concerns. The collection doesn't need to know anything of the view. The collection merely dispatches its events during the normal course of its work—blissfully unaware that the view is desperate for its notifications so that it can redraw itself immediately.

8.3 What's Next

Dart exposes a rich eventing system that runs the gamut of developer needs. The event system implemented by Events, XMLHttpRequestEvents, and others is

easy to use and intuitive. And, when these simpler mechanisms are not enough, Dart makes it easy to define our own event systems.

In [Chapter 14, *Futures and Isolates*, on page 115](#), we will discuss another means for separate chunks of code to communicate. In both cases, a little ceremony goes a long way toward keeping code well factored and maintainable.

Part III

Code Organization

With our first taste of Dart's power, it is time to check out something truly unique to Dart: the library system. Previously, we treated the MVC library that we are building as if we were still limited to JavaScript. That is, we put everything into one large file. Dart comes with a sophisticated built-in library system. As we will see, this means that writing large libraries is not only possible but easy.

Project: Extracting Libraries

Back in [Chapter 6, *Project: MVC in Dart*, on page 39](#), we rewrote our simple Dart application in an MVC style similar to the venerable Backbone.js. As is, there is little possibility for reuse of this code—either in our own codebase or shared with others.

In this chapter, we will factor those MVC classes into reusable libraries. This involves two separate activities: putting our newfound object-oriented Dart skills to use and making use of Dart’s excellent library system. The end results will facilitate both code reuse as well as better code maintainability.

Also in this chapter, we will run into something that we won’t find in most language books: some actual limitations of the language being discussed.

9.1 What to Extract and What to Leave

For each of the collection, model, and view classes from [Project: MVC in Dart](#), we are now faced with the question of what to extract.

Collections: Everything but the Hard Stuff

As we did in [Project: MVC in Dart](#), we start with the core of the client-side MVC library: the collection. Since collections are loose code mappings to a REST-like backend, we ought to be able to extract much of `ComicsCollection` into a `HipsterCollection` superclass. Anything that is specific to comics books, such as the `/comics` URL, can stay in `ComicsCollection`. The rest (ideally) can move out to be reused with other REST-like back ends.

With everything else in `HipsterCollection`, `ComicsCollection` can be expressed as follows:

```
class ComicsCollection extends HipsterCollection {
  // url => the url root of the collection
  // other comics book specific methods, if any
}
```

If we move everything in `HipsterCollection`, our starting point looks like this:

`mvc_library/collection_skel.dart`

```
class HipsterCollection implements Collection {
  var on, models;

  // constructor
  HipsterCollection() {
    on = new CollectionEvents();
    models = [];
  }

  // TODO: get URL from subclass
  // MVC
  fetch() { /* ... */ }
  create(attrs) { /* ... */ }
  add(model) { /* ... */ }

  // Collection
  void forEach(fn) { /* ... */ }
  int get length() { /* ... */ }
  operator [](id) { /* ... */ }
}
class CollectionEvent implements Event { /* ... */ }
class CollectionEvents implements Events { /* ... */ }
class CollectionEventList implements EventListenerList { /* ... */ }
```

Aside from renaming the constructor to `HipsterCollection()`, very little else needs to change. In fact, the only changes that are required involve the URL, which we planned for, and the `create()` method, which can no longer hard-code new `ComicsBook()` in order to generate models.

Our first instinct for enabling the subclass to communicate the URL to `HipsterCollection` is to set a property in the superclass for the implementation to define.

```
class HipsterCollection implements Collection {
  var url;
  // ....
}
```

The subclass could then define the url.

```
// This won't work!!!
class ComicsCollection extends HipsterCollection {
  var url = '/comics';
}
```

Unfortunately, this will not work. In Dart, instance variables defined in the body of the base class are not available to the superclass. If the instance variable is set in the constructor or another method, then the superclass will see the change. Here we are trying to define it in the body of the class definition. If the `HipsterCollection` base class attempted to access the `url` property, it would not see the property as defined in the concrete class definition of `ComicsCollection`. So, we need to “trick” it with a getter.

`mvc_library/comics_with_url.dart`

```
class ComicsCollection extends HipsterCollection {
  get url() => '/comics';
}
```

Since the `url()` getter is referenced in the `HipsterCollection` superclass but must be defined in a subclass, we declare it as abstract.

`mvc_library/collection_with_abstract_url.dart`

```
class HipsterCollection implements Collection {
  abstract String get url();
  // ...
}
```

This allows `fetch()` to work without change. The `url()` getter looks like just another instance variable within the class.

With URL out of the way, let’s turn our attention to a mechanism for the concrete class to tell the superclass how to build models. This turns out to be trickier than the `url()` getter.

In Backbone.js, for instance, the class that creates models is conveyed by a property.

`mvc_library/backbone_sub_class.js`

```
var Comics = Backbone.Collection.extend({
  model: ComicBook
});
```

This will not work in Dart because classes are not first-order objects. That is, there is no way to assign a class name to a variable or use one as the value of a Hash/Map. So, we have to settle for a factory method that, given model attributes, will return a new instance of the model that we desire. That is, we add a `modelMaker()` method.

`mvc_library/comics_with_model_maker.dart`

```
class Comics extends HipsterCollection {
  get url() => '/comics';
  modelMaker(attrs) => new ComicBook(attrs);
}
```


Back in the `HipsterCollection` superclass, we again declare this as abstract and update `create()` to use this method.

```
mvc_library/collection_with_abstract_model_maker.dart
class HipsterCollection implements Collection {
  abstract HipsterModel modelMaker(attrs);
  // ...
  create(attrs) {
    var new_model = modelMaker(attrs);
    new_model.save(callback:(event) {
      this.add(new_model);
    });
  }
}
```

Both the `url()` getter and the `modelMaker()` method serve as examples of something that is easy in JavaScript being more difficult in Dart. Neither of these difficulties is written in stone—at some point relatively soon, Dart may support easier implementations for one or both use cases.

The reason for the limitation is a simple question of priorities. The Dart designers favored defining a well-structured, classical, object-oriented paradigm over treating classes as first-order objects. They favored strongly encapsulated instance variables over shared definitions between classes. And their choices seem to be well-supported given that our “workarounds” were one-liners.

The Model: Nothing to See Here

The implementation of the `HipsterModel` class is even simpler—everything goes there, leaving a subclass to do nothing other than redirecting the constructor.

```
mvc_library/model_comic_book.dart
class ComicBook extends HipsterModel {
  ComicBook(attributes) : super(attributes);
}
```

Again, we have the bother of explicitly passing the subclass constructor arguments to the superclass. Aside from that, the `HipsterModel` base class takes care of everything (recall that the `url` comes from the collection).

If instances of `HipsterModel` are ever used directly, then our subclass would need to override the `urlRoot` getter.

```
mvc/model_url_root.dart
class ComicBook extends HipsterModel {
  // ...
  get urlRoot() => 'comics';
}
```

This is not to suggest that HipsterModel is simple. It is still responsible for updating and deleting records, as well as generating events for which collections and views can listen. Still, there are no surprises or complications when we extract the code out of ComicBook and put it in HipsterModel.

Views and Post-initialization

One of the shortcomings of the optional constructor syntax is that we need to explicitly delegate it in subclasses.

`mvc/view_sub_class_constructor.dart`

```
class Comics extends HipsterView {
  Comics([collection, model, el]):
    super(collection:collection, model:model, el:el);
}
```

OK, it is not *that* much of a bother, but it would be nice if future versions of Dart could shrink that to a single line.

Views need to be able to listen for model and collection events, but this is a very subclass-dependent definition. To accommodate this, we call a `post_initialize()` method from the constructor.

`mvc/view_sub_class_constructor2.dart`

```
class HipsterView {
  // ...
  HipsterView([el, this.model, this.collection]) {
    this.el = (this.el is Element) ? el : document.query(el);
    this.post_initialize();
  }
  void post_initialize() { }
}
```

The `post_initialize()` method might seem like it would best be defined as a private method, but Dart does not allow access from a superclass to a subclass's private methods.

Important: In Dart, all private methods are exclusively private to the library in which they were defined.

With that, we can define our Comics view as follows:

`mvc/view_sub_class_constructor3.dart`

```
class ComicsViews {
  // ...
  ComicsView([this.el, this.model, this.collection]) {
    _subscribeEvents();
    _attachUiHandlers();
  }
}
```

The private `_subscribeEvents()` and `_attachUiHandlers()` methods are the same from [Project: MVC in Dart](#); they are event handlers that render the comic book collection on the screen and delegate UI event handlers for deleting comic books from the collection. In both cases, they are specific to our Dart Comics application, not the Hipster MVC library.

9.2 Real Libraries

Our `main.dart` is getting awfully crowded at this point. We have the `main()` entry point, `HipsterCollection` (and associated event classes), `HipsterModel` (and associated event classes), `HipsterView`, and the various concrete classes. Besides being too large for ease of maintainability, how are we going to realize any reusability?

To solve both problems, we move classes out into separate files. We will use Dart's built-in library support to make this transition both smooth and well positioned for future reuse and maintainability.

Starting with `HipsterCollection`, let's create `HipsterCollection.dart`. To make this a proper Dart library, we need to start with the `#library()` directive. We also need to `#import()` the necessary core packages explicitly required by `HipsterCollection`.

```

mvc_library/collection_library.dart
#library('hipster_collection');

#import('dart:html'); // For events
#import('dart:json');

class HipsterCollection implements Collection {
  // Hip Collection stuff goes here...
}

```

When we pull the `ComicsCollection` class out into its own `ComicsCollection.dart` file, it too needs an opening `#library()` statement. It also needs to pull in `HipsterCollection` so that it can subclass it. Defining a subclass of the `HipsterCollection` base class is a matter of extending `HipsterCollection` and defining those two abstract methods. It should look something like this:

```

mvc_library/collection_comics.dart
#library('my comic book collection');

#import('HipsterCollection.dart');
#import('Models.ComicBook.dart');

class Comics extends HipsterCollection {
  get url() => '/comics';
  modelMaker(attrs) => new ComicBook(attrs);
}

```

Note: I find it best to use a bit of Hungarian notation in my MVC filenames (for example, `Models.ComicBook.dart` and `Collections.ComicBook.dart`), but *not* in the class names themselves because that can make code very noisy.

We defer the details of `#library()` until the next chapter. It is extremely powerful. Our last word on the matter in this chapter will be to look at what happens to the `main.dart` entry point after everything else is pulled out into separate library files.

`mvc_library/main.dart`

```
#import('Collections.Comics.dart', prefix: 'Collections');
#import('Views.Comics.dart', prefix: 'Views');
main() {
  var my_comics_collection = new Collections.Comics()
    , comics_view = new Views.Comics(
      el: '#comics-list',
      collection: my_comics_collection
    );

  my_comics_collection.fetch();
}
```

We instantiate a comics collection and pass that and the DOM ID of the unordered list to the view constructor. Finally, we fetch the collection and allow the various events to trigger the view to render itself when appropriate.

Note the prefix option on the `#import()` statements. To keep the code as clean as possible, both the Comics view collection and the Comics collection proper were defined with a class name of Comics.

```
// Collections.Comics.dart
class Comics extends HipsterCollection { /* ... */}
```

```
// Views.Comics.dart
class Comics extends HipsterView { /* ... */ }
```

It would seem overly wordy to declare the view, for instance, as `ComicsView` extends `HipsterView`. But, when both are used in the same context, there is the very real problem of colliding class definitions. This is where Dart's prefix option for `#import()` is extremely handy.

By importing `Collections.Comics.dart` with the `Collections` prefix, all top-level class definitions are now referenced with the `Collections.*` prefix. That is, to instantiate a collection object, we use `new Collections.Comics()`. That is a huge help with code organization, especially when working with applications that define numerous libraries (as is typical in MVC applications).

Note: For the curious, the final version of our Hipster MVC library is located at <https://github.com/eee-c/hipster-mvc/>.

9.3 What's Next

We put our object-oriented knowledge from [Classes and Objects](#) to some good use here. Along the way, we exposed a few of Dart's warts. It is not possible, for instance, to pass class names as we would variables. It is not possible to define an instance variable in a subclass so that the superclass can see it. It is not possible for a superclass to access private methods of a subclass. Even though some of these realities may fly in the face of what we might expect in object-oriented code, Dart has some good reasons for this, as we will explore in upcoming chapters.

Regardless of the restrictions, we also found some fairly unobtrusive workarounds. And by workarounds, I mean "The Dart Way." Better still, our object-oriented refactoring put us in a good position to make use of Dart's very cool library system.

Code reuse and maintainability are seemingly impossible challenges to master in the browser, and yet Dart handles libraries with ease. By factoring our MVC library and classes out into their own files, we make it easy to find and maintain specific aspects of our codebase. At the same time, we have sacrificed none of the ease of working with the code; aside from the introduction of a few `#import()` and `#library()` statements, our code is unchanged from when it was all in one big file.

In the next chapter, we will take a look at libraries in a little more depth and then discuss `#source()`, the cousin to `#import()`. When we pick back up with our project in [Chapter 11, Project: Varying Behavior, on page 91](#), we will adapt it for use with local storage.

Libraries

JavaScript has been around for seventeen years. In all of that time, it still lacks a simple library loading mechanism. This is not for lack of need. There are many independent solutions and even several attempts at standards (commonjs,¹ AMD,² and ECMAScript harmony modules³). As these standards have languished in various states of usefulness and adoption, the community has generated more loader plugins than can be counted.

Despite all of these efforts, the surest way to load additional JavaScript libraries is via a combination of additional `<script>` tags and compressing multiple files into a single compressed JavaScript script. Neither solution is without problems (such as load order and deployment complexity).

Mercifully, Dart has the concept of libraries built in. Better yet, they are very easy to work with. Dart currently supports two different vehicles for importing functionality into Dart code: `#source()` and `#import()`.

10.1 #source()

The `#source()` directive is used to include arbitrary chunks of Dart code into the current context. For instance, we might have a pretty-printing function along with some local variables stored in `pretty_print.dart`.

libraries/pretty_print.dart

```
var INDENT = '  ';
pretty_print(thing) {
  if (thing is List) print(INDENT + Strings.join(thing, ', '));
  else print(thing);
}
```

1. <http://www.commonjs.org/>
2. <https://github.com/amdjs/amdjs-api/wiki/AMD>
3. <http://wiki.ecmascript.org/doku.php?id=harmony:modules>

Nothing special is needed in the source file to allow it to be included in another file. To actually include it, we simply need to supply the path to the source file. The path can be either relative or absolute.

```
libraries/print_things.dart
```

```
#source('pretty_print.dart');

main() {
  var array = ['1', '2', '3']
    , hash = {'1': 'foo', '2': 'bar', '3': 'baz'}
    , string = "Dart is awesome";

  pretty_print(array);
  pretty_print(hash);
  pretty_print(string);
}
```

This would produce the following output:

```
1, 2, 3
Instance of 'LinkedHashMapImplementation'
Dart is awesome
```

If we wanted to reuse this nifty little printer without copying and pasting, we use the same `#source()` statement in any code file that needs it.

Limitations

The temptation with `#source()` is to use it to mix a collection of methods into a class. Since Dart lacks multiple inheritance, this would be a nice way to work around that limitation. Unfortunately, the `#source()` statement (like the `#import()` statement) must be declared at the top of the source file.

Thus, the following would not work:

```
class Circle {
  #source('pretty_print.dart');
  var x,y;

  Circle.pretty(this.x, this.y) {
    pretty_print(this);
    pretty_print(x);
    pretty_print(y);
  }
}
```

There are ways to get around this limitation using `noSuchMethod`, but they are awkward.⁴

4. <http://japhr.blogspot.com/2012/01/dart-mixins.html>

10.2 #import()

Of more interest is the `#import()` statement, which allows us to import classes for use in our code. From the start, Dart supports this feature that comes standard in all server-side languages. Better still, it works transparently in the browser.

Unlike the `#source()` statement, the source file that is being imported requires a single modification: the `#library()` statement must exist at the top of the file. Consider, for example, a pretty-printing stopwatch class that might be used to time code.

```
libraries/pretty_stop_watch.dart
#library('A very pretty stop watch class');

class PrettyStopwatch {
  Stopwatch timer;
  PrettyStopwatch() {
    timer = new Stopwatch();
  }
  PrettyStopwatch.start() {
    timer = new Stopwatch.start();
  }
  start() {
    timer.start();
  }
  stop() {
    timer.stop();
    print("Elapsed time: " + timer.elapsedInMs() + "ms");
  }
}
```

The string in the `#library()` statement is used in the output of the `dartdoc` documentation and in the comments of compiled JavaScript. It also serves as a nice (required) minimal form of documentation.

If we wanted to time how long it took to count to ten million, we would use our pretty stopwatch thusly:

```
libraries/time_counting.dart
#import('pretty_stop_watch.dart');

main() {
  var timer = new PrettyStopwatch.start();
  for(var i=0; i<10000000; i++) {
    // just counting
  }
  timer.stop();
}
```


As nice as the `#source()` statement is, the `#import()` statement has much more potential for helping to organize our code. Not only is Dart strongly object-oriented, but it makes it dirt easy to share and reuse class libraries, even in the browser.

Note: Chrome is smart enough to load `#source()` and `#import()` files only once, no matter how many different places they might be referenced.

Prefixing Imports

The `#import()` statement allows us to namespace imported classes. Even though we typically import only the classes that we need directly in Dart code, there is still the very real possibility for name collision.

Consider, for example, the case in which our MVC application needs the Comics collection as well as the Comics view. Both would be declared as class Comics (although they would extend `HipsterCollection` and `HipsterView`). If we attempt to import them directly, Dart's compiler throws an already-defined exception.

To get around this potential limitation, we prefix the imports.

`libraries/prefixed_imports.dart`

```
#import('collections/Comics.dart', prefix: 'Collections');
#import('views/Comics.dart', prefix: 'Views');
#import('views/AddComic.dart', prefix: 'Views');
```

With that, we no longer reference the Comics view class or the Comics collection class. Instead, we use `Views.Comics` and `Collections.Comics`.

`libraries/using_prefixes.dart`

```
main() {
  var my_comics_collection = new Collections.Comics(),
      comics_view = new Views.Comics(
        el: '#comics-list',
        collection: my_comics_collection
      );
}
```

The implication of prefixes is that there is no global object namespace in Dart. The `main()` entry point has its own, isolated workspace with its own classes and objects. The various libraries that are imported all have their own object namespace. This cuts down on much of the ceremony involved with code organizing and is another way that Dart encourages us to write clean code.

10.3 Core Dart Libraries

Dart defines a set of core libraries, the documentation for which is always publicly available at <http://api.dartlang.org>. At the time of this writing, the core libraries include `dart:core`, `dart:coreimpl`, `dart:isolate`, `dom`, `html`, `io`, `json`, `uri`, and `utf`. Each library defines a number of common classes that we might want to make use of in our applications.

To use one of these core libraries, we use the `#import()` statement just like we would do for our own defined libraries.

```
#import('html');  
#import('json');
```

Packaging with “pub”: Although still in the very early stages of development, Dart already includes the pub packaging tool. Ultimately this tool will evolve to the point at which it can install packages from a central repository. Already, it is capable of resolving and installing dependencies. An early example of using this tool for local web development is located at <http://japhr.blogspot.com/2012/05/dart-pub-for-local-development.html>.

10.4 What's Next

The built-in ability to organize code is a significant win for Dart. The light-weight syntax that Dart employs ensures that we no longer have an excuse for messy client-side code.

Part IV

Maintainability

Hot on the heels of learning how to organize Dart code, it is time to take a look at some strategies for keeping code maintainable. First up, we update the Hipster MVC library to accommodate multiple methods of syncing data with a remote (or even local) back end. Then, we look at one of Dart's newer features: testing.

Project: Varying Behavior

Those of us coming from a dynamic language background expect to be able to perform all manner of crazy hackery at runtime. Not satisfied with changing a response based on state, we like to change implementation.

In JavaScript, for instance, it is possible to replace the method on an object's prototype at any time. In Ruby, nothing prevents us from replacing a function with a lambda or a Proc. We revel in metaprogramming and cry foul when newbies look at it as magic.

In Dart, there are far fewer opportunities for magic. But it is still possible. To explore this topic, we again return to our comic book catalog application. This time, we will replace the Ajax back-end calls with in-browser storage.

11.1 Vary Class Behavior with `noSuchMethod()`

We first met `noSuchMethod()` in [Chapter 7, *Classes and Objects*, on page 53](#). Let's try to put it to use as a means for switching the behavior of the `save()` method in the Hipster MVC library.

Recall that when `HipsterModel` invokes `save()`, it sends a JSON representation of its attributes to the REST-like data store and establishes handlers for successful updates.

[varying_the_behavior/xhr_save.dart](#)

```
class HipsterModel {  
  // ...  
  save([callback]) {  
    var req = new XMLHttpRequest()  
    , json = JSON.stringify(attributes);  
  
    req.on.load.add((event) {  
      attributes = JSON.parse(req.responseText);  
    });  
  }  
}
```

```

        on.save.dispatch(event);
        if (callback != null) callback(event);
    });

    req.open('post', '/comics', true);
    req.setRequestHeader('Content-type', 'application/json');
    req.send(json);
}
}

```

To successfully replace this with a local storage implementation, we need to save locally, ensure that the same callbacks are called, and ensure that the same events are dispatched. To save a model in `localStorage`, we overwrite an in-memory copy of the database with the new or updated model and save the entire database.

varying_the_behavior/local_save.dart

```

class HipsterModel {
  // ...
  save([callback]) {
    var id, event;

    if (attributes['id'] == null) {
      attributes['id'] = hash();
    }

    id = attributes['id'];
    collection.data[id] = attributes;

    window.
      localStorage.
        setItem(collection.url, JSON.stringify(collection.data));

    event = new Event("Save");
    on.save.dispatch(event);
    if (callback != null) callback(event);
  }
}

```

We will worry about the details of `localStorage` later. For now, we are left in a situation in which we have manually replaced the Ajax implementation with a `localStorage` version of `save`. This is not a long-term recipe for success in getting others to use our library.

One possible solution would be to create a `LocalHipsterModel` subclass of `HipsterModel` containing this new behavior. This is an unsatisfactory solution in that it requires us, as library authors, to build and maintain subclasses for any number of storage behaviors that our users might want: Ajax, `localStorage`,

indexed DB, web sockets, and so on. Worse yet is that it is nearly impossible to anticipate all of the idiosyncrasies that each of these behaviors might need.

Instead of suffering through a series of subclasses, we can use `noSuchMethod()`. Recall from [Classes and Objects](#) that `noSuchMethod()` is a last resort for Dart if it is unable to locate the invoked method anywhere else.

Dart invokes `noSuchMethod()` with two arguments: the name of the method being invoked and the list of arguments being supplied to the method. The first thing that any `noSuchMethod()` implementation should do is guard for the known methods that it is capable of handling.

```
class HipsterModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }
    // ...
  }
}
```

In this case, if anything other than the `save()` method has been invoked, we immediately throw an exception signaling that there is no such method.

Warning: There is currently no way to implement more than one `noSuchMethod()` in a class ancestry chain. If the `ComicBook` model uses `noSuchMethod()` to vary the saving algorithm, then the `noSuchMethod()` in the `HipsterModel` base class will never be seen. This severely restricts when `noSuchMethod()` can legitimately be used. Specifically, it should be used only in concrete classes that will never be extended.

With the guard in place, we are ready to invoke either the local storage `save` or the Ajax `save`. The naïve approach would be to pass the arguments directly to the two private methods that hold this behavior.

```
// *** THIS WON'T WORK ***
class HipsterModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }
    if (useLocal()) {
      _localSave(args);
    }
    else {
      _ajaxSave(args);
    }
  }
}
```

This fails, however, because `_localSave()` and `_ajaxSave()` expect parameters. In the previous implementation, we are passing a List. Instead, we need to manually extract the arguments and place them in the appropriate parameter position.

```
// *** THIS STILL WON'T WORK ***
class HipsterModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }

    if (useLocal()) {
      _localSave(args[0]);
    }
    else {
      _ajaxSave(args[0]);
    }
  }
}
```

There is *still* a problem with the previous code. The `HipsterModel`'s `save()` method can be invoked with an optional callback parameter.

```
new_model.save(callback:(event) {
  this.add(new_model);
});
```

The `noSuchMethod()` method ignores the optional parameter label. Instead, it sees the callback parameters as the first argument. Since `HipsterModel#save()` accepts only a single argument, this does not matter.

Important: Optional parameters in declared methods can be placed in any order and still have the same meaning: `new Comics(el: '#comics', collection: my_comics)` is the same as `new Comics(collection: my_comics, el: '#comics')`. Methods built dynamically by `noSuchMethod()` are restricted to being called in a specific order. The implication is that `noSuchMethod()` can be used only for fixed arity methods.

Thus, our dynamic `save()` can be implemented as follows:

```
class HipsterModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }
    if (useLocal()) {
      _localSave(callback: args[0]);
    }
  }
}
```

```

    else {
      _ajaxSave(callback: args[0]);
    }
  }
}

```

As we have seen, the use cases for `noSuchMethod()` are limited to concrete classes with fixed arity arguments. There is also an implied limitation of being restricted to a single class. In the case of our MVC library, we need a mechanism to synchronize data in both models *and* collections.

What we need is a class that describes this syncing behavior and that allows developers to inject different behavior as needed.

11.2 Sync Through Dependency Injection

The `noSuchMethod()` can be a fairly powerful tool in Dart, but it is limited to a single class or interface. In this case, we need a mechanism to inject a syncing behavior that can be shared between model and controller.

As we saw in [Classes and Objects](#), Dart is fairly well locked down against doing such things. But there is a way.

Let's create a `HipsterSync` class that holds our data syncing behavior. Ultimately, the various libraries that rely on `HipsterSync` will invoke a static method `HipsterSync.call()` to dispatch the CRUD operation. Before looking at that, however, we need a default behavior that can perform Ajax requests.

[varying_the_behavior/hipster_sync_default.dart](#)

```

#library('Sync layer for HipsterMVC');
#import('dart:json');

class HipsterSync {
  static _defaultSync(method, model, [options]) {
    var req = new XMLHttpRequest();
    _attachCallbacks(req, options);
    req.open(method, model.url, true);

    // POST and PUT HTTP request bodies if necessary
    if (method == 'post' || method == 'put') {
      req.setRequestHeader('Content-type', 'application/json');
      req.send(JSON.stringify(model.attributes));
    }
    else {
      req.send();
    }
  }
}

```


That all looks fairly normal now that we have taken the initial Ajax-based app and converted it to an MVC framework. We create an XMLHttpRequest object, open it, and then send the request. New here is the need to support passing request bodies with POST and PUT requests, but a simple conditional suffices to cover this behavior.

All of the classes that will use this sync need to be able to dispatch events upon successful load of the XMLHttpRequest object. The `_attachCallbacks()` static method takes care of this for us.

`varying_the_behavior/hipster_sync_default_callbacks.dart`

```
class HipsterSync {
  static _default_sync(method, model, [options]) {
    var req = new XMLHttpRequest();
    _attachCallbacks(req, options);
    // ...
  }

  static _attachCallbacks(request, options) {
    if (options == null) return;
    if (options.containsKey('onLoad')) {
      request.on.load.add((event) {
        var req = event.target,
            json = JSON.parse(req.responseText);

        options['onLoad'](json);
      });
    }
  }
}
```

This `_attachCallbacks()` method lets us rewrite `HipsterModel#save()` with an `onLoad` callback passed via `options`.

`varying_the_behavior/hipster_model_save_with_sync.dart`

```
class HipsterModel {
  // ...
  save([callback]) {
    HipsterSync.call('post', this, options: {
      'onLoad': (attrs) {
        attributes = attrs;

        var event = new ModelEvent('save', this);
        on.load.dispatch(event);
        if (callback != null) callback(event);
      }
    });
  }
  // ...
}
```

With that, we have delegated data syncing to `HipsterSync`. The first two arguments to `HipsterSync.call()` instruct the sync that it should POST when syncing and that the current model should be used to obtain the serialized data to be sent to the back-end store.

At this point, we are finally ready to look at `HipsterSync.call()`. As we might expect, if no alternative sync strategy has been supplied, it invokes a `_defaultSync()`.

varying_the_behavior/hipster_sync_call.dart

```
class HipsterSync {
  static call(method, model, [options]) {
    if (_injected_sync == null) {
      return _defaultSync(method, model, options:options);
    }
    else {
      return _injected_sync(method, model, options:options);
    }
  }
  // ...
}
```

The interesting behavior is that `_injected_sync()` beastie. It may look like another static method, but it is, in fact, a class variable. User libraries can inject behavior into this library via a `sync=` setter, which expects a function.

varying_the_behavior/hipster_sync_injected.dart

```
class HipsterSync {
  static var _injected_sync;
  static set sync(fn) {
    _injected_sync = fn;
  }
  static call(method, model, [options]) {
    if (_injected_sync == null) {
      return _defaultSync(method, model, options:options);
    }
    else {
      return _injected_sync(method, model, options:options);
    }
  }
  // ...
}
```

The injected function will need to accept the same arguments that `_defaultSync()` does.

Warning: It would make more sense to have a `sync` setter *and* a `sync` class method. Unfortunately, Dart will throw an “already defined” internal error if a method is declared with the same name as a setter. Hence, we need to declare a `sync=` setter and a `call` static method.

With all of this in place, let's switch our HipsterSync strategy to localStorage. This can be done back in the main() entry point for the application. For now, we restrict ourselves to supporting only the GET operations.

varying_the_behavior/main_with_local_sync.dart

```
#import('HipsterSync.dart');

main() {
  HipsterSync.sync = localSync;

  // Setup collections and views ...
}

localSync(method, model, [options]) {
  if (method == 'get') {
    var json = window.localStorage.getItem(model.url),
        data = (json == null) ? {} : JSON.parse(json);

    if (options is Map && options.containsKey('onLoad')) {
      options['onLoad'](data.getValues());
    }
  }
}
```

That is pretty nifty. With a single line, it is possible to inject completely different data syncing behavior for the entire framework.

It is worth pointing out that, because of how Dart manages libraries, setting Hipster.sync in our main.dart file, like so:

```
// main.dart
#import('HipsterSync.dart');

main() {
  HipsterSync.sync = localSync;
  //
}
```

will affect the `_injected_sync` HipsterSync class variable that is seen by HipsterModel.

```
// HipsterModel.dart
#library('Base class for Models');

#import('HipsterSync.dart');

class HipsterModel {
  // I see HipsterSync._injected_sync from main.dart
}
```

And the same goes, of course, for HipsterCollection.

```
// HipsterCollection.dart
#library('Base class for Collections');

#import('HipsterSync.dart');

class HipsterCollection {
  // I see HipsterSync._injected_sync from main.dart
}
```

Each of these files, `main.dart`, `HipsterModel.dart`, `HipsterCollection.dart`, and `HipsterSync.dart`, are separate files. And yet Dart ensures that the `HipsterSync` class defined in one is the same that is seen by all. The only equivalent in JavaScript is what Backbone.js does to define its `Backbone.sync`. Backbone declares a global variable (for example, `Backbone`) and instructs developers that it needs to be included via `<script>` tag before all other code. Using something like `require.js` will get you close to Dart's behavior, but it is very nice to have this working at the outset of the language rather than attempting to tack it on eighteen years after the fact.

11.3 What's Next

As mentioned in [Classes and Objects](#), Dart frowns on dynamic language features that are necessary for metaprogramming. Even so, it is quite possible to achieve some pretty nifty dynamic language features in Dart. The `noSuchMethod` method is certainly an easy one to hook into for a significant portion of metaprogramming. It is limited to instance methods, but this ought to cover 80 percent of a developer's dynamic programming needs. When that fails, there are still ways to exploit Dart's functional nature to achieve broader dynamic language features.

That said, there are still limitations to what can be done dynamically in Dart. Once the reflection and mirroring part of the specification is complete, Dart should be a dynamic language programmer's dream.

Testing Dart

Note: This chapter describes “bleeding-edge” functionality. Just about every aspect of Dart is in flux, but the test harness used here has not even made the jump into a regular state of flux. Even so, it is an important enough topic that it warranted inclusion.

As web applications grow in complexity, it is no longer sufficient to rely on type checking to catch bugs. In this chapter, we will explore testing the Hipster MVC library, which has definitely grown to the too-complex-for-type-checking point.

12.1 Obtaining the Test Harness

As of early 2012, the best place to get the Dart testing harness is the “bleeding edge” branch of the Dart subversion repository. The following command will export the test harness to the tests/lib directory of the current working directory:

```
SVN_HOST=http://dart.googlecode.com  
SVN_PATH=/svn/branches/bleeding_edge/dart/client/testing  
  
svn export ${SVN_HOST}${SVN_PATH} tests/lib
```

With that, we are ready for testing.

12.2 2 + 2 = 5 Should Be Red

Interestingly, the Dart testing harness can, and is intended to be, run directly in our application. Its output will pop up over the normal display of the application. This is invaluable for instant gratification/assurance that the build is still green while development is ongoing. That said, we will start with a dummy test page and Dart code.

Any old HTML will suffice as long as it does the following:

- Pulls in the Dart code to be tested
- Starts the Dart engine

The dummy test page to test the HipsterCollection class will be the following:

testing/tests/01.html

```
<html>
<head>
  <title>Hipster Test Suite</title>
  <script type="application/dart"
    src="HipsterCollectionTest.dart"></script>
  <script type="text/javascript">
    // start Dart
    navigator.webkitStartDart();
  </script>
</head>
<body>
<h1>Test!</h1>
</body>
</html>
```

We rely on HipsterCollectionTest.dart to import two required testing libraries as well as our own source code. It also needs to declare the main() entry point since we have not declared it elsewhere.

testing/tests/HipsterCollectionTest.dart

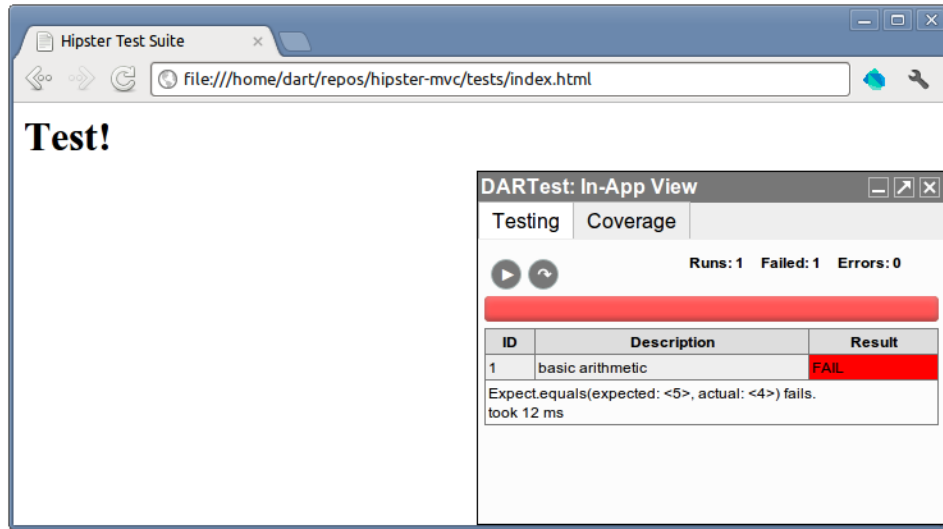
```
#import('lib/unittest/unittest_darttest.dart');
#import('lib/darttest/darttest.dart');
#import("../public/scripts/HipsterCollection.dart");
main() {
  // Tests go here!
  new DARTest().run();
}
```

In addition to the imports, we create a new DARTest instance and start the suite with the run() method. Nothing will happen without a test, however, so let's write one. And in the grand tradition of Behavior-Driven Development, let's start with a failing test.

testing/tests/02test.dart

```
#import('lib/unittest/unittest_darttest.dart');
#import('lib/darttest/darttest.dart');
#import("../public/scripts/HipsterCollection.dart");
main() {
  test('basic arithmetic', (){
    Expect.equals(5, 2 + 2);
  });
  new DARTest().run();
}
```

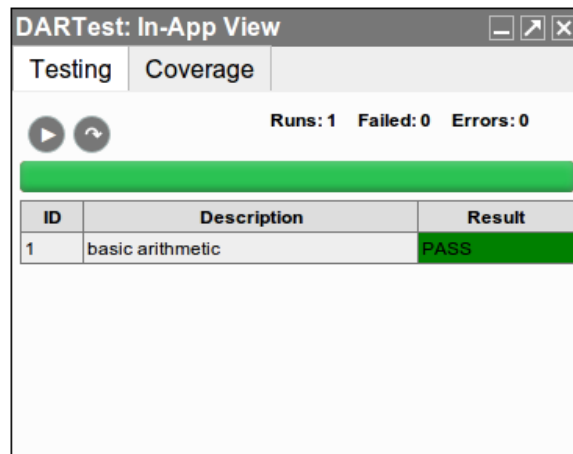
To run this test, we load the HTML in the browser, click the Run button in the DARTest: In-App View pop-up, and are greeted with the following:



Yay! A failing test! To get the test to pass, we simply need to fix our math.

```
test('basic arithmetic', (){
  Expect.equals(5, 2 + 3);
});
```

Reloading the pages produces a green test suite.



Note: The page needs to be reloaded each time the code or test is changed. This reflects that the test suite is meant to be live in the real application, not on dummy pages. The purpose behind this is so that developers can see the actual application and the test suite at all times.

Now that we have a basic idea of how to write tests, let's replace our silly arithmetic test with a real test of the `HipsterCollection` class.

```
test('HipsterCollection has multiple models', () {
  HipsterCollection it = new HipsterCollection();
  it.models = [{id: 17}, {id: 42}];

  Expect.equals(2, it.length);
});
```

This is a simple test of the length getter in our `HipsterCollection` class. The `test()` function takes two arguments: a string describing the test and an anonymous function that includes at least one expectation. After a bit of setup, the expectation is checked using Dart's built-in `Expect.equals()` test.

In addition to the basic `Expect.equals` test, Dart also supports numerous convenience methods ranging from approximation (`Expect.approxEquals`) to basic type checks (`Expect.isNull`, `Expect.isFalse`, `Expect.MapEquals`). It even supports checking for exceptions. For instance, we can verify that `HipsterCollection#fetch()` fails without a URL.

```
test('HipsterCollection fetch() fails without a url', () {
  HipsterCollection it = new HipsterCollection();
  Expect.throws(() {it.fetch();});
});
```

Despite the youth of the test harness, it already has facilities for grouping common behavior in tests. For instance, we might want to describe two aspects of `HipsterCollection` lookup.

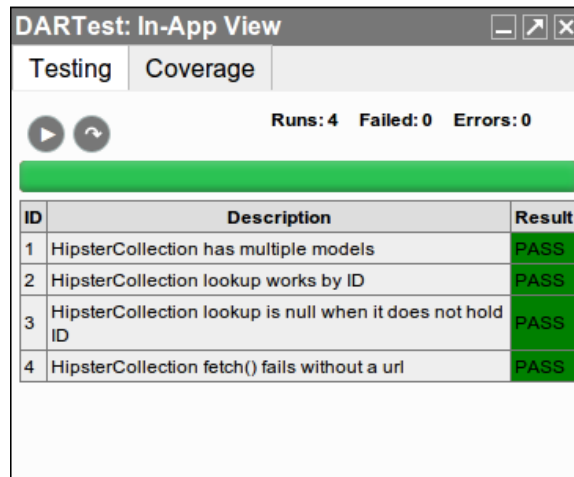
```
group('HipsterCollection lookup', () {
  var model1 = {id: 17},
      model2 = {id: 42};

  HipsterCollection it = new HipsterCollection();
  it.models = [model1, model2];

  test('works by ID', () {
    Expect.listEquals([17], it[17].getValues());
    Expect.listEquals(['id'], it[17].getKeys());
    Expect.equals(model1, it[17]);
  });

  test('is null when it does not hold ID', () {
    Expect.isNull(it[1]);
  });
});
```

Just like that, we have a test suite with four passing tests.



The screenshot shows a window titled "DARTest: In-App View" with two tabs: "Testing" and "Coverage". Below the tabs, there are two circular icons (a play button and a refresh button) and the text "Runs: 4 Failed: 0 Errors: 0". A green progress bar is visible above a table of test results.

| ID | Description | Result |
|----|---|--------|
| 1 | HipsterCollection has multiple models | PASS |
| 2 | HipsterCollection lookup works by ID | PASS |
| 3 | HipsterCollection lookup is null when it does not hold ID | PASS |
| 4 | HipsterCollection fetch() fails without a url | PASS |

Asynchronous Testing

Dart, like JavaScript, is a functional language. Functional languages present unique challenges to testing. The built-in expectations will not help us much here, but the testing harness does. The tool of choice in this case is the `asyncTest` function, which allows us to set expectations for how often a callback will be invoked.

Consider, for instance, adding a new element to a `HipsterCollection`. The expectation in this case is that any listeners for the insert events will be invoked. If we have one listener, then we expect one callback to be invoked. This can be expressed as follows:

```
asyncTest('HipsterCollection add dispatch add event', 1, () {
  // test goes here...
})
```

The `asyncTest()` function introduces a new, second argument in between the test description and the test function. This second argument is the number of times that callbacks will be invoked. The rest of the `asyncTest` setup is the same as plain-old `test()` calls.

To note when a callback has actually been invoked, the testing harness supplies the `callbackDone()` function. Every time this function is called, an internal counter is incremented. As long as this internal counter and the number declared in `asyncTest()` agree at the end of the test run, the test is considered to have passed.

To test the insert event, we add a `callbackDone()` listener to the `on.insert` list of listeners and then add a new element to the collection.

```

asyncTest('HipsterCollection add dispatches insert events', 1, () {
  // don't care about data sync in this case
  noOpSync(method, model, [options]) {}
  HipsterSync.sync = noOpSync;

  HipsterCollection it = new HipsterCollection();

  it.
    on.
    insert.
    add((event) {
      callbackDone();
    });

  it.add({'id': 42});
});

```

With that, we have five reasonably useful tests in place to help catch regressions in our HipsterCollection class.

12.3 What's Next

Although still quite new (which is saying something for a language as new as Dart), the test suite shows significant promise. Not only does it already support a wealth of testing primitives, but it supports the sometimes difficult task of callback testing. That the test suite can run on top of applications should help developers spot regressions significantly faster than would be possible in a separate test suite.

Part V

The Next Level with Dart

At this point, we have a strong understanding of the power of Dart under our collective belts. With that, it is time to begin discussing what comes next as Dart evolves. First, we remove the last vestiges of JavaScript thinking from Hipster MVC by removing callbacks. Callbacks—long the bane of many a JavaScripter—are replaced with completers, futures, and isolates that promise a simpler way to describe what happens later. Last, we talk about Dart's support for HTML5 and where Dart goes from here.

Project: An End to Callback Hell

One of the knocks on many large JavaScript codebases is the inevitable tangle of callbacks scattered throughout. There is definite power in keeping execution frames small. We need look no further than the rise of Node.js for evidence to support this. But even the most experienced JavaScripters can be confused by the interplay of a myriad of callbacks and events. So far, our Hipster MVC library exhibits many of the same characteristics of the JavaScript approach—and we have yet to even attempt error handling! As we saw in [Chapter 8, Events, on page 67](#), the syntax for Dart events is different from in JavaScript, but the approach is very much the same. This is not the case with callbacks. Let's take a look at how the Future can significantly improve the long-term maintainability of Dart applications.

13.1 The Future

When last we saw our HipsterModel, we had replaced direct Ajax calls with a data synchronization layer that was cleverly named HipsterSync. In the save() method, this looks like this:

```
class HipsterModel {  
  // ...  
  save([callback]) {  
    HipsterSync.call('post', this, options: {  
      'onLoad': (attrs) {  
        attributes = attrs;  
        var event = new ModelEvent('save', this);  
        on.load.dispatch(event);  
        if (callback != null) callback(event);  
      }  
    });  
  }  
  // ...  
}
```

There are at least three issues with this approach. First, we are invoking `call()` with too many arguments—we need to indicate that this is a “post,” and this is necessary so that `HipsterSync` knows what to sync—but the `options` parameter describes only a side effect and is not needed in order to perform the main execution thread. Second, the readability of the `onLoad` callback is less than ideal, buried inside the `options` parameter. Third, it is weak programming practice to null check for the presence of a callback each time we attempt to invoke the callback.

Instead of the callback approach, let’s switch to `Future` objects. In fact, `Futures` are little more than objects that formalize callbacks. If we switch `HipsterSync.call()` to return a `Future`, then we can inject a callback via the `then()` method.

```
class HipsterModel {
  // ...
  save([callback]) {
    HipsterSync.
      call('post', this).
      then((attrs) {
        this.attributes = attrs;
        on.load.dispatch(new ModelEvent('save', this));
        if (callback != null) callback(event);
      });
  }
  // ...
}
```

With that small change, the intention of `HipsterSync.call()` is much clearer; it does nothing more than POST this (the model) to the back-end data store. Once that is complete, *then* we grab the attributes returned from the data store to do the following:

- Update the model’s attributes
- Notify any listeners that the load event is complete
- Invoke the callback if it is present

We are in better shape, but there is still the matter of the null check. There are times that a conditional like the one in the `then()` statement is necessary. More often than not, a null check is our code begging for a better abstraction. This is one of those times.

Instead of an optional callback, we can convert `save()` to return a `Future`. The easiest way to generate a `Future` is by instantiating a new `Completer`, which has a future getter that can be returned.

```
Future<HipsterModel> save() {
    Completer completer = new Completer();
    // ...
    return completer.future;
}
```

Here, we explicitly state that our Future will return a value of HipsterModel by declaring the return type of save() to be Future<HipsterModel>. That is, upon successful save, save() will send a copy of the current model back to the then() function. For instance, if we wanted to log the ID of a newly created model, we could use the then() function thusly:

```
var comic_book = new ComicBook({'title': 'Batman'});
comic_book.
  save().
  then((new_comic) {
    print("I got assigned an ID of ${new_comic['id']}");
  });
```

We still need to tell save() how to actually notify the then() statement in the calling context that anything happened. This is done with the same Completer object that produced the Future. To notify the Future that the Completer has completed, we invoke complete() with the value to be sent to then.

```
class HipsterModel {
    // ...
    Future<HipsterModel> save() {
        Completer completer = new Completer();
        HipsterSync.
            call('post', this).
            then((attrs) {
                this.attributes = attrs;
                on.load.dispatch(new ModelEvent('save', this));
                completer.complete(this);
            });
        return completer.future;
    }
    // ...
}
```

When HipsterSync.call() successfully completes, then we update the model's attributes, dispatch its events, and mark save() as complete. By completing the Completer, code that calls save() can then() do what they need to do. This almost begins to read like English, which is nice.

More importantly, we have significantly improved the readability, and hence the maintainability, of the save() method. We no longer have to worry about an optional callback parameter to save(). Now, invoking call() on HipsterSync involves

only the two things needed to effect the requested change (the action to take and the model). There is no more options Map to clutter things. Lastly, by converting this method to a Future itself, we have eliminated a conditional, and in doing so, we have improved life for the caller.

13.2 Handling Errors in the Future

Until now, we have conveniently ignored the question of how to handle exceptions. What happens if our POST to the back end results in a 400-class error? How can we design Hipster MVC so that developers can handle exceptions appropriately?

Had we stuck with options, the answer would have been to add yet another callback inside the Map of options. Luckily for us, Completer and Future have a formal mechanism for dealing with just such a situation. A Completer invokes `completeException()` to signal a problem and Future deals with problems with `handleException()` to do something with that exception.

The default data sync behavior in `HipsterSync` would signal an exceptional condition to the future by calling `completeException()` when the request status is not OK (for example, greater than 299).

```
class HipsterSync {
  // ...
  static Future _defaultSync(method, model) {
    var request = new XMLHttpRequest(),
        completer = new Completer();
    request.
      on.
      load.
      add((event) {
        var req = event.target;
        if (req.status > 299) {
          completer.
            completeException("That ain't gonna work: ${req.status}");
        }
        else {
          var json = JSON.parse(req.responseText);
          completer.complete(json);
        }
      });
    // Open and send the request
    return completer.future;
  }
}
```

The value in `completeException` does not have to be a subclass of `Exception`—any old object will do. In this case, we supply a simple string.

```
if (req.status > 299) {
  completer.
    completeException("That ain't gonna work: ${req.status}");
}
```

Back in the model class, we need to handle this exceptional case. The `then()` method returns `void`, so it is not chainable. This requires us to store the after-call `Future` back in `HipsterModel.save()` in a local variable. We can then inject a happy-path `then()` behavior into the `Future` as well as an `handleException()` behavior.

```
class HipsterModel {
  // ...
  Future<HipsterModel> save() {
    Completer completer = new Completer();
    Future after_call = HipsterSync.call('post', this);

    after_call.
      then((attrs) { /* ... */ });

    after_call.
      handleException((e) { /* ... */ });
    return completer.future;
  }
}
```

Since `HipsterModel#save()` is itself a `Future`, it should handle exceptions with a `completeException()` of its own.

```
after_call.
  handleException((e) {
    completer.completeException(e);
    return true;
  });
```

13.3 What's Next

As we will see in [Chapter 14, *Futures and Isolates*, on page 115](#), `Futures` come in handy in other places. It is easy to see why. Through a very basic application in the Hipster MVC library, we have significantly improved the maintainability of the library as well as made it easier for developers to use the library.

Having built-in objects codifying this behavior is a big win for Dart. It is not hard to build something similar in JavaScript. Even so, built-in `Futures` allow us to focus on writing beautiful code, not on writing code that allows us to write beautiful code.

Futures and Isolates

In addition to the familiar syntax and concepts that Dart supports, there are a number of newer features that Dart brings to the table. Among them are a class of higher-level functional programming features that include the likes of completers, futures, and isolates.

14.1 Completers and Futures

Completers are objects that encapsulate the idea of finishing a task at some later point. Rather than passing a callback to be invoked along with a future value, Dart allows us to define the entire thing in a completer object.

Since completers trigger an action in the future, completers are intimately tied to Futures. The single most defining characteristic of a Future is a callback function, which is supplied via a Future object's `then()` method.

In its simplest form, we can create a Completer object and grab the future property from the completer so that we can specify what happens when the completer finishes. Finally, some time later, we tell the completer that it is finished with an optional message.

```
main() {  
  var completer = new Completer();  
  
  var future = completer.future;  
  future.then((message) {  
    print("Future completed with message: " + message);  
  });  
  completer.complete("foo");  
}
```

The end result of this is printing out the following message:

Future completed with message: foo

On their own, Futures provide an important but narrow functionality. They do one thing but do it well.

Completers can be completed only once. Trying it twice will throw an error. Here's an example:

```
completer.complete("foo");
completer.complete("bar");
```

This will produce the following:

```
Future completed with message: foo
Unhandled exception:
Exception: future already completed
```

In addition to sending successful messages from the future, it is possible to signal errors. To support error handling, the Future needs to define a `handleException()` callback.

```
main() {
  var completer = new Completer();

  var future = completer.future;
  future.handleException((e) {
    print("Handled: " + e);
    return true;
  });

  var exception = new Exception("Too awesome");
  completer.completeException(exception);
}
```

When the completer completes with an exception, the result is as follows:

```
Handled: Exception: Too awesome
```

If the `handleException()` callback does not return `true`, the exception is not caught. Rather, it would continue to bubble up the call stack until caught elsewhere or until the application is forced to terminate.

14.2 Isolates

As the name implies, Dart isolates are used to isolate long-running functions from the main thread of execution. Isolates use Futures to signal to the main thread that they are ready for doing things.

```
main() {
  final doom = new DoomsDay();
  print('Certain doom awaits...');
  doom.spawn().then((port) {
```

```

var year = 2012;
port.call({'year':year}).receive((message, replyTo) {
    print("Doom in $year is on a $message.");
});
});
}

```

We will take a look at the DomsDay isolate in a bit. For now, recognize that the spawn() call returns a Future. Once the isolate is running, the Future completes, invoking the then() callback with a message port for communication. We can call it with an argument and await messages on the receive callback.

The doomsday algorithm used in the DomsDay isolate class is rather simple. It finds the day of the week for the last day in February. With simple mnemonics, we can use that to figure out the day of the week for any day in the year.¹ It makes for a small algorithm that is useful for illustration.

Any isolate class needs to implement the Isolate interface. Only one method is required, main().

```

class DomsDay extends Isolate {
  main () {
    // heavy-duty calculation here
  }
}

```

The ability for an isolate to be spawned and to return an on-ready Future is already baked into the Isolate interface. What we need to do in main() is establish the message port on which we can receive the calling context's messages. That is, when the calling context sends a message via call(), like so:

```

port.call({'year':year}).receive((message, replyTo) {
    // ...
});

```

we establish our isolate's ability to receive that message in main() by setting the receive() callback of the port property.

```

class DomsDay extends Isolate {
  main () {
    port.receive((message, replyTo) {
      // Process the incoming message
      // Send a replyTo to the calling context
      // when done
    });
  }
}

```

1. http://en.wikipedia.org/wiki/Doomsday_rule

Just like the `spawn()` method, the `port` property is already defined for us by the `Isolate` interface. Cleverly, an isolate will wait for the `port` to be defined before calling `main()`; thus, we can be assured that defining the `receive()` callback shown previously will actually be defined on a real thing.

The full implementation of our `DoomsDay` class pulls the year out of the incoming message and assigns it to a class variable. We can then reply to the caller with the result of the `day()` method's calculation.

```
class DoomsDay extends Isolate {
  int _year;

  main () {
    port.receive((message, replyTo) {
      _year = message['year'];
      replyTo.send(this.day());
    });
  }

  day() {
    var march1 = new Date(_year, 3, 1, 0, 0, 0, 0)
    , doom = march1.subtract(new Duration(1)).weekday
    , dow = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'];

    return dow[doom];
  }
}
```

The result of calling our `DoomsDay` isolate in the following code:

```
main() {
  final doom = new DoomsDay();

  print('Certain doom awaits...');

  doom.spawn().then((port) {
    var year = 2012;
    port.call({'year':year}).receive((message, replyTo) {
      print("Doom in $year is on a $message.");
    });
  });
}
```

is that 2012's doomsday is a Wednesday.

Certain doom awaits...

Doom in 2012 is on a Wed.

Beware the Wednesday.

14.3 Heavy vs. Light Isolates

The ultimate goal with isolates is to provide a mechanism to run isolates in separate threads (“heavy” isolates) or in the currently executing thread (“light”). At the time of this writing, there is no difference between the two in Dart proper. Interestingly, when isolate code is compiled down to JavaScript, heavy and light isolates do work as expected, with heavy isolates running their own thread—oftentimes much faster than their light isolate brethren.

14.4 Wrapping Up

Completers, Futures, and isolates are a class of solutions that we may use relatively infrequently. Even so, they are definitely used, and it is wonderful to know that we do not have to reinvent them or choose the best library available when we need them.

Recent versions of JavaScript include the concept of web workers, but if we need to support older browsers, we are left to our own devices. Early versions of Node.js supported promises, which are quite similar to Futures in Dart. In the end, they were removed from Node.js, leaving the poor hipster to reinvent promises each time the need arises.

Thankfully, in Dart, web workers and promises are supported from the outset in the form of isolates.

HTML5 and Dart

Back in [Chapter 4, *Manipulating the DOM*, on page 25](#), we saw many examples of how easy Dart makes it to interact and manipulate the DOM and styles. This chapter builds upon basic DOM manipulation to present a guide to adding a little life to web pages through animations, WebSockets, and other sundry techniques that fall under the HTML5 umbrella.

Most of this chapter discusses features that can be accomplished already in JavaScript. What Dart brings to the table is a familiar, simple syntax and cross-platform compatibility (no need for @-webkit and @-moz duplication).

15.1 Animation

If you are making interactive, modern websites in 2012 and beyond, a little animation can go a long way. The transition CSS property is one of those small, seemingly innocuous additions that in reality packs in quite a bit of functionality. Consider, for example, the form view from our comic book application. When it is rendered, it might be nice to fade in.

```
#import('dart:html');
#import('HipsterView.dart');

class AddComicForm extends HipsterView {
  // ...
  render() {
    el.style.opacity = '0';
    el.innerHTML = template();
    window.setTimeout(() {
      el.style.transition = '1s ease-in-out';
      el.style.opacity = '1';
    }, 1);
  }
}
```

The transition property is the same one from CSS3.¹ It is a space-separated string of individual transition properties describing the following:

- To which styles the transition applies (all, opacity, and so on). In our example, this is not included, so it defaults to all.
- Duration of the animation. Our animation lasts for one second.
- The animation function. There are several available including ease, ease-in, ease-out, and linear. The ease-in-out function that we use starts slow, accelerates, and then eases into a slow finish.
- The delay before the animation begins. Since we did not include this, the default of no delay (0s) is used.

Note: When specifying an initial style, a transition, and a finished state, Dart has the habit of “optimizing” away the transition. As a workaround, we place the transition and final state inside a `setTimeout()`. This effectively takes the animation out of the normal synchronous workflow just enough to allow the animation to kick in.

15.2 Local Storage

Dart’s support for client-side storage is still somewhat unsettled, but, as we saw in [Chapter 11, Project: Varying Behavior, on page 91](#), it is far enough along that we can already perform `localStorage`. Although it is synchronous and can be slow, `localStorage` is the most widely supported client-side storage solution—and the only one currently supported by Dart.

Because it is synchronous (that is, its operations block other activity in client-side applications), it is not well suited for large stores of data. Still, it is quite handy for smaller datasets and prototyping. There is a benefit to its synchronous nature: there is far less ceremony involved in using it.

The API for working with `localStorage` is nearly identical to the traditional JavaScript API. In both, it is inefficient to store individual objects of a collection separately. Instead, we store serialized JSON representations of Lists or Maps.

```
var json = window.localStorage.getItem('Comic Books'),
    comics = (json != null) ? JSON.parse(json) : [];
```

Adding a record to `localStorage` is a simple matter of updating the deserialized data, reserializing it, and storing the whole thing back in the database.

1. <https://developer.mozilla.org/en/CSS/-moz-transition>

```
// Oops. We'll fix the spelling in a bit...
comics.push({'id':42, 'title':'Sandmn'});
window.localStorage.setItem(
  'Comic Books',
  JSON.stringify(comics)
);
```

This will replace the data that was previously stored in the Comic Books localStorage item.

Updating data in the local store then consists of nothing more than updating the item in the local representation and serializing that back into the data store as JSON.

```
var json = window.localStorage.getItem('Comic Books'),
    comics = (json != null) ? JSON.parse(json) : [];

comics.forEach((comic_book) {
  if (comic_book['title'] == 'Sandmn') {
    comic_book['title'] == 'Sandman';
  }
});
window.localStorage.setItem(
  'Comic Books',
  JSON.stringify(comics)
);
```

Similarly deleting is accomplished by removing from the local copy and serializing that back into the localStorage item.

```
var json = window.localStorage.getItem('Comic Books'),
    comics = (json != null) ? JSON.parse(json) : [];

awesome_comics.filter((comic_book) {
  return (comic_book['id'] >= 42);
});
window.localStorage.setItem(
  'Comic Books',
  JSON.stringify(comics)
);
```

It does not get much easier than localStorage. Unfortunately, each of those operations blocks the browser from doing anything else. So, if our application has too much data on the browser, then it is time to consider something with a little more power.

Important: At the time of this writing, the APIs for both IndexedDB and Web SQL were not ready for regular usage. Ideally, this will change in time for the 1.1 edition of this book.

15.3 WebSockets

WebSockets is a wonderful new technology that allows for truly asynchronous communication between the browser and the server. No longer are web developers relegated to awkward hacks like comet or Ajax long-polling. The browser can now open a websocket to the server and push data over that open connection on demand. Better still, when the server has new information available, it can push it immediately to the user over that same websocket.

Dart's support for WebSockets is quite nice. This makes it nearly trivial to, for example, swap out the data syncing layer in our comic book app to use WebSockets instead of Ajax.

```
#import('dart:html');
#import('HipsterSync.dart');

// Library scope so that both main() and wsSync()
// have access to the same websocket
WebSocket ws;
main() {
  HipsterSync.sync = wsSync;
  ws = new WebSocket("ws://localhost:3000/");

  // Don't fetch until the websocket is open so
  // that wsSync can talk over an active
  // websocket
  ws.
    on.
    open.
    add((_) {
      var my_comics_collection = new Collections.Comics()
      my_comics_collection.fetch();
      // other initialization...
    });
}
```

We create websocket objects by instantiating the `WebSocket` constructor with a proper `ws://` websocket URL. Websockets are completely asynchronous, which includes opening the connection. Therefore, we add a listener for the websocket's open event. When the connection is open, we can start performing data synchronization operations such as fetching the data over the websocket.

Note: At the time of this writing, Dart does not support declaring subprotocols for websockets.

Sending messages over websockets is trivial—we need only invoke `ws.send(message)`. Recall that the data sync method in `HipsterSync` needs to accept two arguments: the CRUD method and the model (or collection) being synced.

Using that information, we can craft a message to be sent over a websocket to the back end.

```
wsSync(method, model) {
  String message = "$method: ${model.url}";
  String message = "$method: ${model.url}";
  if (method == 'delete')
    message = "$method: ${model.id}";
  if (method == 'create')
    message = "$method: ${JSON.stringify(model.attributes)}";

  ws.send(message);
}
```

That will send the message, but we need to handle a response from the server and, in turn, inform the rest of the stack of the response. As we saw in [Chapter 13, *Project: An End to Callback Hell*, on page 109](#), informing the HipsterSync class is done with a Future. So, our wsSync layer needs its own Completer object, and it needs to complete once the response has been received from the server.

```
wsSync(method, model) {
  final completer = new Completer();
  String message = /* determine the message */

  ws.send(message);
  ws.
    on.
      message.
        add(_wsHandler(event) {
          completer.
            complete(JSON.parse(event.data));
          event.target.on.message.remove(_wsHandler);
        });

  return completer.future;
}
```

The return value of our sync function is a Future. HipsterSync expects this and, in turn, has a corresponding then() clause to propagate this information throughout the Hipster MVC stack once the completer has been marked as finished. Upon receipt of the server response inside the message handler, we complete the Future with the message from the server, which is available in the message event's data attribute.

In this case, we want to listen only for a single response from the server, so we remove the handler after the completer is finished. If this message handler had been left in place, it would continue to receive messages upon the second

response from the server (for example, in response to a user-initiated create). Since the completer referenced inside this sync closure has already been completed, the application would generate all sorts of messages about already completed completers.

Websockets in Dart are really no more difficult or easier than their counterpart in JavaScript. Still, there is a uniquely Darty take on it, which makes them a pleasure to use.

15.4 Canvas

Dart lacks something like Raphaël.js² that eases some of the pain associated with working with the <canvas> element. Even so, it brings its own Darty take on the staple of HTML5 games everywhere.

As with traditional canvas, Dart still requires a <canvas> element and a corresponding drawing context. If the page already has a <canvas> element, we obtain a drawing context with the getContext() method.

```
CanvasElement canvas = document.query('canvas');
CanvasRenderingContext context = canvas.getContext('2d');
```

With the context, we can then draw all sorts of wonderful things. By way of example, we can draw an empty, white rectangle on the entire canvas as a backdrop.

```
int width = context.canvas.width,
    height = context.canvas.height;

// start drawing
context.beginPath();

// clear drawing area
context.clearRect(0,0,width,height);
context.fillStyle = 'white';
context.fillRect(0,0,width,height);

// done drawing
context.closePath();
```

A plain white background is not terribly interesting. To spice it up a little, we can add a simple red square that will represent our current location in a game room. If our current location is encapsulated by a Player object that has an x and y position, then our initial placement might look something like this:

2. <http://raphaeljs.com/>

```

// start drawing
context.beginPath();

// clear drawing area
// ...
// draw me
context.rect(me.x, me.y, 20, 20);
context.fillStyle = 'red';
context.fill();
context.strokeStyle = 'black';
context.stroke();

// done drawing
context.closePath();

```

That will draw a rectangular me that is 20 pixels in size, filled in with red, and with a black border. All that is needed at this point is a document listener to handle arrow key presses. When an arrow key is pressed, the event can tell the player to move by invoking `move()` in the appropriate direction and redraw the entire canvas.

```

document.
  on.
    keyDown.
      add((event) {
        String direction;

        // Listen for arrow keys
        if (event.keyCode == 37) direction = 'left';
        if (event.keyCode == 38) direction = 'up';
        if (event.keyCode == 39) direction = 'right';
        if (event.keyCode == 40) direction = 'down';
        if (direction != null) {
          event.preventDefault();
          me.move(direction);
          draw(me, context);
        }
      });

```

Here, the `draw()` function performs the same context manipulation that we did previously, only with an updated position for me.

As Dart evolves, no doubt there will be more improvements to the API to make it a little easier with which to work. More importantly, there ought to be many libraries built on top of it. Already there is an early port of the Box2D library into Dart that can draw simple physics. Aptly named `DartBox2D`,³ it is well worth checking out.

3. <http://code.google.com/p/dartbox2d/>

15.5 Wrapping Up

This was rather a grab bag of various tools currently available to the Dart developer. But really, that is what HTML5 is—a grab bag of techniques available in newer browsers. They are techniques handled with aplomb by Dart. Perhaps by the time that the 1.1 edition of this book is ready, each of these sections can be broken out into their own chapters. As good as it is now, it is only going to improve as Dart itself continues to evolve.

And Dart is going to evolve. Dramatically. Nearly 100 people are dedicated full-time to improving the language. More importantly, the community that has sprung up around Dart is extremely active. So, please, join us at <https://groups.google.com/a/dartlang.org/group/misc/topics>, and add your voice to the community!

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

<http://pragprog.com/titles/csdart>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: <http://pragprog.com/titles/csdart>

Contact Us

Online Orders: <http://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://pragprog.com/write-for-us>

Or Call: +1 800-699-7764