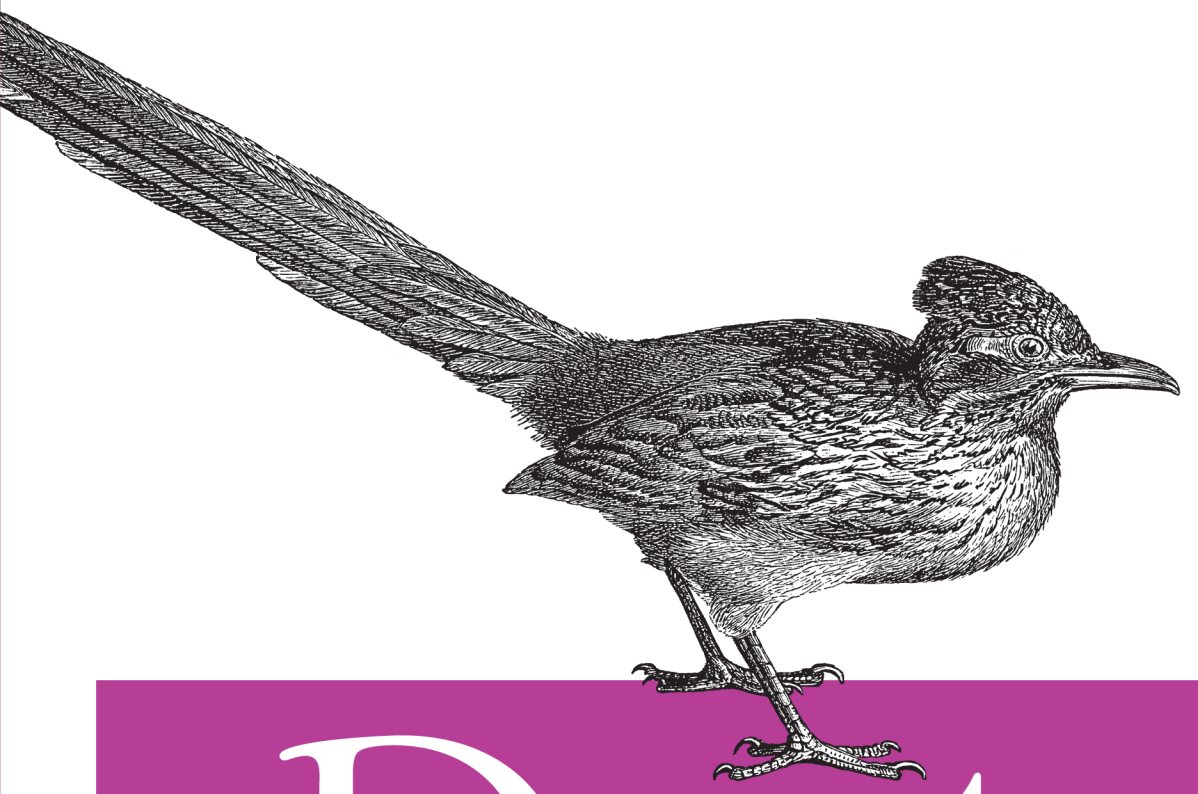
A New, Tool-Friendly Language for Structured Web Apps



Dart

Up and Running

O'REILLY® | Google™ PRESS

Kathy Walrath
& Seth Ladd

Dart: Up and Running

Kathy Walrath and Seth Ladd

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Dart: Up and Running

by Kathy Walrath and Seth Ladd

Printed in the United States of America.

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

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

Editor: Meghan Blanchette
Production Editor: Christopher Hearse
Proofreader: Christopher Hearse
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

Revision History for the First Edition:

2012-10-24 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449330897* for release details.

ISBN: 978-1-449-33089-7

[LSI]

Table of Contents

Foreword

When we joined Google and entered the fascinating world of web browser development more than six years ago, the web was a different place. It was clear that a new breed of web apps was emerging, but the performance of the underlying platform left much to be desired. Given our background in designing and implementing virtual machines, building a high performance JavaScript engine seemed like an interesting challenge. It was. We implemented the V8 JavaScript engine from scratch and shipped it as part of Google Chrome in 2008, and we are very proud of the positive performance impact our work seems to have had on the entire browser industry.

Even though recent performance gains in web browsers have shattered most limits on how large and complex web apps can be, building large, high-performance web apps remains hard. Without good abstraction mechanisms and clean semantics, developers often end up with complex and convoluted code. Naturally, this problem gets exacerbated as the codebase grows. We designed the Dart programming language to solve this exact problem, and we hope that programmers will be more productive as a result.

Over the past year, we have read and written a lot of Dart code, and it is very satisfying to see how Dart inspires programmers to strive for concise, elegant programs. There is something very enjoyable about incrementally transforming prototypes into maintainable production software through refactorings and adding type annotations—and it definitely feels like Dart as a language scales well from small experiments to large projects with lots of code.

Dart: Up and Running is a practical guide that introduces the Dart programming language and teaches you how to build Dart applications. We hope you will enjoy the book and Dart.

—Lars Bak and Kasper Lund
Designers of the Dart programming language, October 2012

Preface

You don't need to be an expert web developer to build web apps. With Dart, you can be productive as you build high-performance apps for the modern web.

Our aim for this book is to be a useful introduction to the Dart language, libraries, and tools. Because this book is short and Dart is young, you might also need to refer to the Dart website at *http://dartlang.org*—both for details and for updates. For the latest news, keep an eye on the Dart page on Google+.

Another important website is this book's GitHub project. The text for this work is available there under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. Source code files for this book's samples are also there, in the `code/` subdirectory. Downloading the sample code from GitHub is much easier than copying it from the book.

If you find an error in the sample code or text, please create an issue.

Conventions Used in This Book

The following typographical conventions are used in this book:

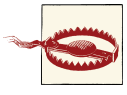
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

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

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

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

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



This icon indicates a warning or caution.

Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Dart: Up and Running* by Kathy Walrath and Seth Ladd (O'Reilly). Copyright 2013 Kathy Walrath and Seth Ladd, 978-1-449-33089-7."

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

Safari® Books Online

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

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

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database

from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

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

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

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

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

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

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

Acknowledgments

We'd like to thank the many people who contributed to this book. We hope we haven't forgotten anyone, but we probably have.

The following Dart engineers and managers gave us prompt, helpful reviews of the sections corresponding to their areas of responsibility: Mads Ager, Peter von der Ahé, Justin Fagnani, Dan Grove, Florian Loitsch, Sam McCall, Vijay Menon, John Messerly, Anton Muhin, Lasse R.H. Nielsen, Bob Nystrom, Ivan Posva, and Jaime Wren.

We'd especially like to thank the people who reviewed even bigger swaths of the book or contributed in other, large ways:

- JJ Behrens, whose careful look at the first draft of the book helped us catch errors and inconsistencies, as well as rework Chapter 5 to be more interesting, and less of a laundry list. He also created a system for testing our samples, so we can be sure that they work now and continue to work as the language and libraries evolve.
- Shailen Tuli, who helped test our examples, although he doesn't even work for Google.
- Mary Campione, whose stream-of-consciousness review of the entire book, performed while she was first learning the language, helped us find and fix many confusing spots, as well as some errors.
- Phil Quitslund, who did a big-picture review of the book and gave us guidance and encouragement.
- Kasper Lund, whose review caught issues that only someone with his expert, comprehensive knowledge of the Dart language and libraries could have found.
- Gilad Bracha, the language spec writer whose reviews of the language chapter (our longest one) were invaluable for getting language details right. We couldn't cover everything, so we look forward to his future work on making all the corners of the language understandable to all Dart programmers.

Other Googlers helped, as well. Vivian Li, the head of Chrome Developer Relations, supported our work on this book. Andres Ferrate, the Google Press liaison to O'Reilly, helped simplify the process of getting the book published. Myisha Harris gave us excellent legal advice.

The people at O'Reilly were extremely helpful. Meghan Blanchette, our editor, kept everything going smoothly, monitoring our progress in the nicest possible way. Christopher Hearse checked our work and helped us make some last-minute fixes that improved the final result. We'd also like to thank the good people who manage the author workflow and make working on an O'Reilly book such a pleasure. We personally worked with Sarah Schneider and Jessica Hosman.

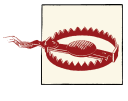
Finally, we thank Lars Bak and Kasper Lund for writing the foreword, and most of all for creating Dart.

CHAPTER 1

Quick Start

Welcome to Dart, an open-source, batteries-included developer platform for building structured HTML5 web apps. This chapter tells you why Google created Dart, what's cool about Dart, and how to write and run your first Dart app.

Dart provides not only a new language, but libraries, an editor, a virtual machine (VM), a browser that can run Dart apps natively, and a compiler to JavaScript. Dart aims to be a more productive way to build the high-performance, modern apps that users demand.

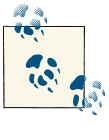


Dart is still changing! This book reflects the Milestone 1 release (October 2012), which aims to finalize the language but not the libraries. Wherever possible, this book tells you what we expect to change.

Why Google Created Dart

Google cares a lot about helping to make the web great. We write a lot of web apps, many of them quite sophisticated—think Gmail, Google Calendar, Google+, and more. We want web apps to load quickly, run smoothly, and present engaging and fun experiences to users. We want developers of all backgrounds to be able to build great experiences for the browser.

As an example of Google's commitment to the web, consider the Google Chrome browser. Google created it to spur competition at a time when the web platform seemed to be stagnating. It worked. As Figure 1-1 shows, browser speed has increased immensely since Chrome's introduction in 2008.



The JavaScript engine known as *V8* is responsible for much of Chrome's speed. Many of the V8 engineers are now working on the Dart project.

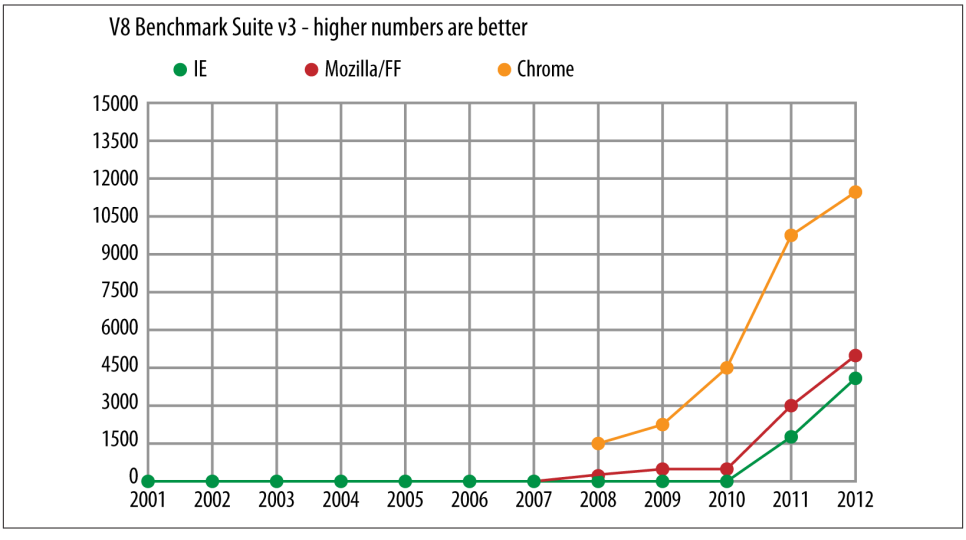


Figure 1-1. Browser speed

The number of new features in browsers has also increased, with APIs such as WebGL, FileSystem, Web workers, and WebSockets. Browsers now have automatic update capabilities, frequently delivering new features and fixes directly to the user. Mobile devices such as tablets and phones also have modern browsers with many HTML5 features.

Despite these improvements in the web platform, the developer experience hasn't improved as much as we'd like. We believe it should be easier to build larger, more complex web apps. It's taken far too long for productive tools to emerge, and they still don't match the capabilities offered by other developer platforms. You shouldn't have to be intimately familiar with web programming to start building great apps for the modern web. And even though JavaScript engines are getting faster, web apps still start up much too slowly.

We expect Dart to help in two main ways:

- *Better performance:* As VM engineers, the designers of Dart know how to build a language for performance. A more structured language is easier to optimize, and a fresh VM enables improvements such as faster startup.
- *Better productivity:* Support for libraries and packages helps you work with other developers and easily reuse code from other projects. Types can make APIs clearer and easier to use. Tools help you refactor, navigate, and debug code.

A Quick Look at the Dart Language

It's hard to talk about a language without seeing it. Here's a peek at a small Dart program:

```
import 'dart:math';

class Point {
  num x, y;
  Point(this.x, this.y);
  num distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}

main() {
  var p = new Point(2, 3);
  var q = new Point(3, 4);
  print('distance from p to q = ${p.distanceTo(q)}');
}
```

Of course, Dart's main use case is building modern web apps. Programming the browser is easy:

```
import 'dart:html';

main() {
  var button = new ButtonElement();
  button..id = 'confirm'
        ..text = 'Click to Confirm'
        ..classes.add('important')
        ..on.click.add((e) => window.alert('Confirmed!'));
  query('#registration').elements.add(button);
}
```

You'll learn about the Dart language and libraries in Chapters 2 and 3 respectively.

What's Cool About Dart

Dart may look familiar, but don't let that fool you. Dart has lots of cool features to help give you a productive and fun experience building the next generation of awesome web apps.

Dart is easy to learn. A wide range of developers can learn Dart quickly. It's an object-oriented language with classes, single inheritance, lexical scope, top-level functions, and a familiar syntax. Most developers are up and running with Dart in just a few hours.

Dart compiles to JavaScript. Dart has been designed from the start to compile to JavaScript, so that Dart apps can run across the entire modern web. Every feature considered for the language must somehow be translated to performant and logical JavaScript before it is added. Dart draws a line in the sand and doesn't support older, legacy browsers.

Dart runs in the client and on the server. The Dart virtual machine (VM) can be integrated into a web browser, but it can also run standalone on the command line. With built-in library support for files, directories, sockets, and even web servers, you can use Dart for full end-to-end apps.

Dart comes with a lightweight editor. You can use Dart Editor to write, launch, and debug Dart apps. The editor can help you with code completion, detecting potential bugs, debugging both command-line and web apps, and even refactoring. Dart Editor isn't required for writing Dart; it's just a tool that can help you write better code faster.

Dart supports types, without requiring them. You can omit types when you want to move very quickly, aren't sure what structure to take, or simply want to express something you can't with the type system. You can add types as your program matures, the structure becomes more evident, and more developers join the project. Dart's optional types are static type annotations that act as documentation, clearly expressing your intent. Using types means that fewer comments are required to document the code, and tools can give better warnings and error messages.

Dart scales from small scripts to large, complex apps. Web development is very much an iterative process. With the reload button acting as your compiler, building the seed of a web app is often a fun experience of writing a few functions just to experiment. As the idea grows, you can add more code and structure. Thanks to Dart's support for top-level functions, optional types, classes, and libraries, your Dart programs can start small and grow over time. Tools such as Dart Editor help you refactor and navigate your code as it evolves.

Dart has a wide array of built-in libraries. The core library supports built-in types and other fundamental features such as collections, dates, and regular expressions. Web apps can use the HTML library—think DOM programming, but optimized for Dart. Command-line apps can use the I/O library to work with files, directories, sockets, and servers. Other libraries include URI, UTF, Crypto, Math, and Unit test.

Dart supports safe, simple concurrency with isolates. Traditional shared-memory threads are difficult to debug and can lead to deadlocks. Dart's isolates, inspired by Erlang, provide an easier to understand model for running isolated, but concurrent, portions of your code. Spawning new isolates is cheap and fast, and no state is shared. In web apps, isolates even compile to Web workers.

Dart supports code sharing. Traditional web programming workflows can't integrate third-party libraries from arbitrary sources or frameworks. With the Dart package manager (pub) and language features such as libraries, you can easily discover, install, and integrate code from across the web and enterprise.

Dart is open source. Dart was born for the web, and it's available under a BSD-style license. You can find the project's issue tracker and source repository at dart.google-code.com. Maybe you'll submit the next patch?

Up and Running

Now that you know something about Dart, get ready to code! These instructions feature the open-source Dart Editor tool. When you download Dart Editor, you also get the Dart-to-JavaScript compiler and a version of Chromium (nicknamed *Dartium*) that includes the Dart VM.

If you run into trouble installing and using Dart Editor, see Troubleshooting Dart Editor.

Step 1: Download and Install the Software

In this step, you'll install Dart Editor and, if necessary, a Java runtime environment. (To avoid having to modify the PATH environment variable, you can install the JRE under your Dart installation directory, in a subdirectory named `jre`.)

1. Download the Dart Editor ZIP file for your platform from *http://www.dartlang.org/downloads.html*.
2. Unzip the file. The resulting directory, which we'll call your *Dart installation directory*, contains the `DartEditor` executable file and several subdirectories, including a `samples` directory.
3. If you don't already have a Java runtime, download and install it. Dart Editor requires Java version 6 or higher.

Step 2: Launch the Editor

Go to your Dart installation directory, and double-click the **DartEditor** executable file  .

You should see the Dart Editor application window appear, looking something like Figure 1-2.

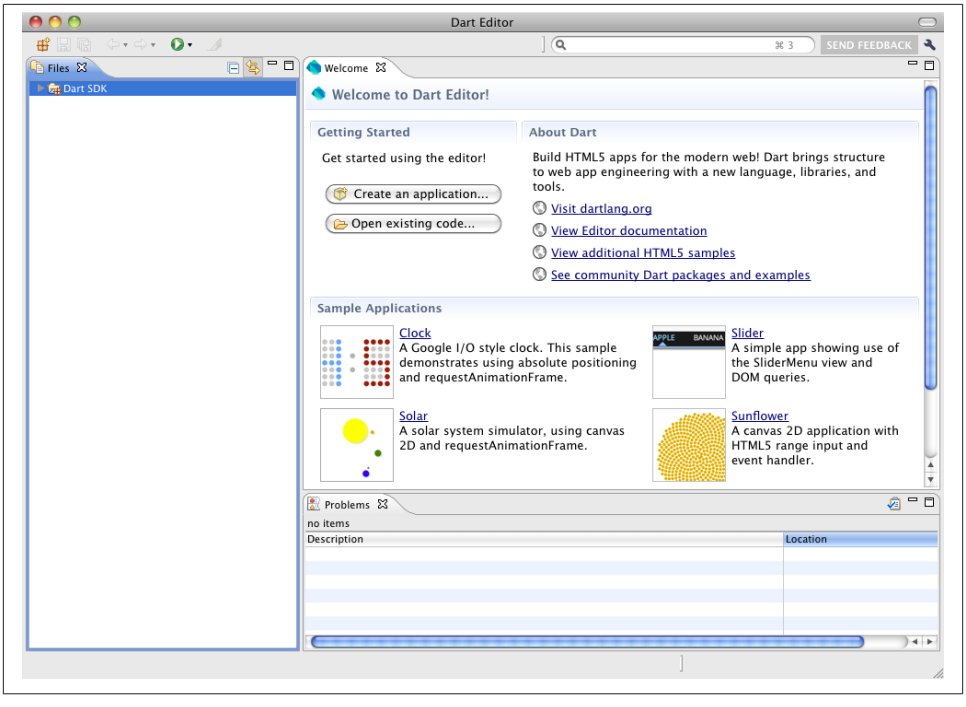


Figure 1-2. Dart Editor and its Welcome page

Step 3: Create and Run an App

It's easy to create a simple web or command-line app from scratch. This step walks you through creating and running a command-line app.

1. Click the New Application button (at the upper left of Dart Editor). Alternatively, choose **File > New Application...** from the Dart Editor menu. A dialog appears (see Figure 1-3).
2. Type in a name for your application—for example, `HelloWorld`. If you don't like the default directory, type in a new location or browse to choose the location.
3. Unselect **Generate content for a basic web app** and **Add Pub support** if they're selected. Then click **Finish** to create the initial files for the app.

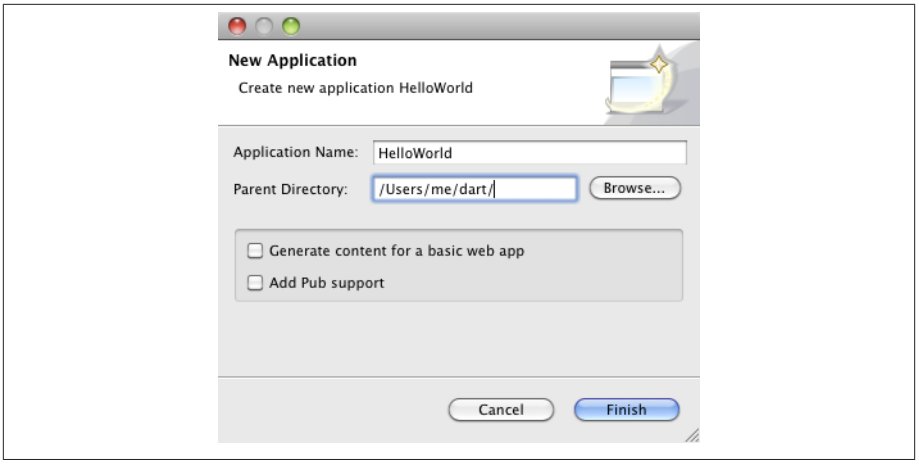


Figure 1-3. Create command-line or web apps with Dart Editor

A default Dart file appears in the Edit view, and its directory appears in the Files view. Your Dart Editor window should look something like Figure 1-4.

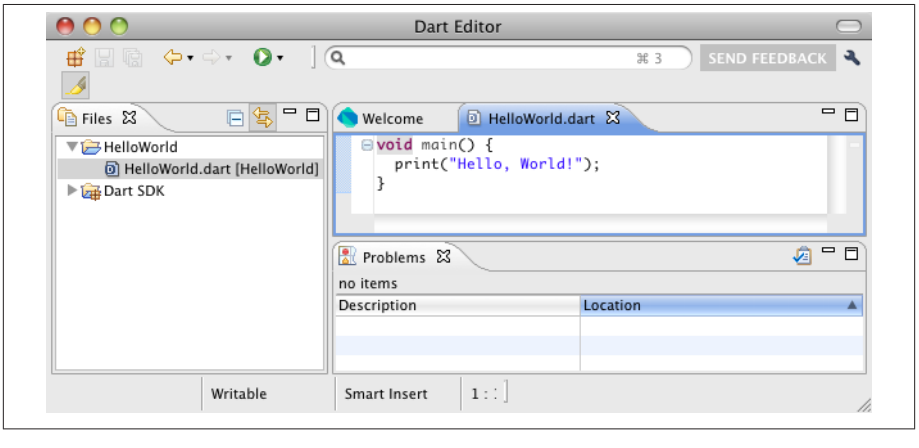


Figure 1-4. Dart Editor displaying a new app's files

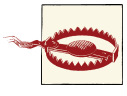
4. Click the Run button to run your new app.

   For command-line apps, the output of `print()` appears at the bottom right, in a new tab next to the Problems tab.

Step 4: Open and Run a Sample

The Dart Editor bundle comes with several samples. In this step, you'll open a sample web app and run it in Dartium.

1. Click the **Welcome** tab. Or choose **Welcome Page** from the **Tools** menu.
2. In the Welcome tab, click the image labeled **Sunflower**. The Editor view now displays the contents of `sunflower.dart`, and the Files view lists the files in the Sunflower app's directory.
3. Click the Run button . Dart Editor launches Dartium, which displays sunflower.html.



Dartium is a technical preview, and it might have security and stability issues. *Do not use Dartium as your primary browser!*

4. Move the slider to display the sunflower, as shown in Figure 1-5. For details about the Sunflower example, read the Sunflower Code Walkthrough.

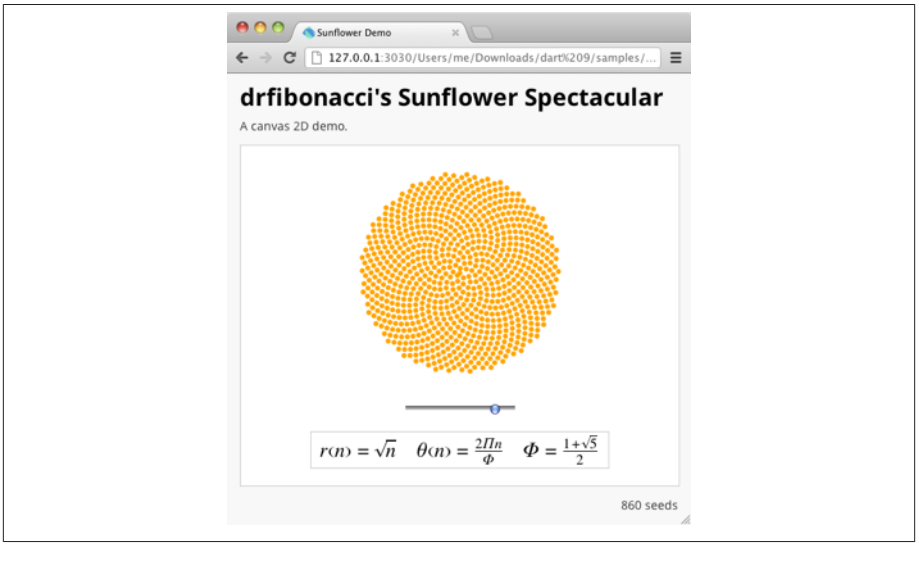


Figure 1-5. The Sunflower sample running in Dartium

What Next?

Now that you know the basics, you can learn more about Dart Editor and help improve it.

Become a power user

See the Dart Editor homepage for help on using Dart Editor's expanding feature set.

Send feedback!

Click the **SEND FEEDBACK** link (at the upper right of the Dart Editor window) whenever you notice a problem or have an idea for improving Dart Editor. We'll open a new issue for you, if appropriate, without disclosing your sensitive or personally identifiable information.

CHAPTER 2

A Tour of the Dart Language

This chapter shows you how to use each major Dart feature, from variables and operators to classes and libraries, with the assumption that you already know how to program in another language.

To play with each feature, create a command-line application project in Dart Editor, as described in "Up and Running" (page 5).

Consult the Dart Language Specification whenever you want more details about a language feature.

A Basic Dart Program

The following code uses many of Dart's most basic features.

```
// Define a function.
printNumber(num aNumber) {
  print('The number is $aNumber.'); // Print to the console.
}

// This is where the app starts executing.
main() {
  var number = 42;           // Declare and initialize a variable.
  printNumber(number);       // Call a function.
}
```

Here's what this program uses that applies to all (or almost all) Dart apps:

`// This is a comment.`
: Use // to indicate that the rest of the line is a comment. Alternatively, use /* ... */. For details, see "Comments" (page 51).

`num`
: A type. Some of the other built-in types are String, int, and bool.

`100`
: A number *literal*. Literals are a kind of compile-time constant.

`print()`
: A handy way to display output.

`'...'` *(or* `"..."`*)*
: A string literal.

`$variableName` *(or* `${expression}`*)*
: String interpolation: including a variable or expression's string equivalent inside of a string literal. For more information, see "Strings" (page 16).

`main()`
: The special, *required*, top-level function where app execution starts.

`var`
: A way to declare a variable without specifying its type.

Our code follows the conventions in the Dart Style Guide. For example, we use two-space indentation.

Important Concepts

As you learn about the Dart language, keep these facts and concepts in mind:

- Everything you can place in a variable is an *object*, and every object is an instance of a *class*. Even numbers and functions are objects. All objects inherit from the Object class.
- Specifying static types (such as `num` in the preceding example) clarifies your intent and enables static checking by tools, but it's optional. (You might notice when you're debugging your code that objects with no specified type get a special type: `dynamic`.)
- Dart parses all your code before running it. You can provide tips to Dart—for example, by using types or compile-time constants—to catch errors or help your code run faster.

- Dart supports top-level functions (such as `main()`), as well as functions tied to a class or object (*static* and *instance methods*, respectively).
- Similarly, Dart supports top-level *variables*, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as *fields* or *properties*.
- Unlike Java, Dart doesn't have the keywords `public`, `protected`, and `private`. If an identifier starts with an underscore (_), it's private to its library. For details, see "Libraries and Visibility" (page 46).
- *Identifiers* can start with a letter or _, followed by any combination of those characters plus digits.
- Sometimes it matters whether something is an *expression* or a *statement*, so we'll be precise about those two words.
- Dart tools can report two kinds of errors: warnings and errors. Warnings are just hints that your code might not work, but they don't prevent your program from executing. Errors can be either compile-time or run-time. A compile-time error prevents the code from executing at all; a run-time error results in an exception (page 32) being raised while the code executes.
- Dart has two *runtime modes*: production and checked. Production is faster, but checked is helpful at development.

Runtime Modes

We recommend that you develop and debug in checked mode, and deploy to production mode.

Production mode is the default runtime mode of a Dart program, optimized for speed. Production mode ignores assert statements (page 32) and static types.

Checked mode is a developer-friendly mode that helps you catch some type errors during runtime. For example, if you assign a non-number to a variable declared as a `num`, then checked mode throws an exception.

Variables

Here's an example of creating a variable and assigning a value to it:

```
var name = 'Bob';
```

Variables are references. The variable called `name` contains a reference to a String object with a value of "Bob".

Default Value

Uninitialized variables have an initial value of `null`. Even variables with numeric types are initially null, because numbers are objects.

```
int lineCount;
assert(lineCount == null);
// Variables (even if they will be numbers) are initially null.
```

The `assert()` call is ignored in production mode. In checked mode, `assert(condition)` throws an exception unless *`condition`* is true. For details, see "Assert" (page 32).

Optional Types

You have the option of adding static types to your variable declarations:

```
String name = 'Bob';
```

Adding types is a way to clearly express your intent. Tools such as compilers and editors can use these types to help you, by providing early warnings for bugs and code completion.

This chapter follows the style guide recommendation of using `var`, rather than type annotations, for local variables.

Final and Const

If you never intend to change a variable, use `final` or `const`, either instead of `var` or in addition to a type. A final variable can be set only once; a const variable is a compile-time constant.

A local, top-level, or class variable that's declared as `final` is initialized the first time it's used.

```
final name = 'Bob';   // Or: final String name = 'Bob';
// name = 'Alice';    // Uncommenting this results in an error
```

Lazy initialization of final variables helps apps start up faster.

Use `const` for variables that you want to be compile-time constants. Where you declare the variable, set the value to a compile-time constant such as a literal, a const variable, or the result of an arithmetic operation on constant numbers.

```
const bar = 1000000;       // Unit of pressure (in dynes/cm2)
const atm = 1.01325 * bar; // Standard atmosphere
```

Built-in Types

The Dart language has special support for the following types:

- numbers
- strings
- booleans
- lists (also known as *arrays*)
- maps

You can initialize an object of any of these special types using a literal. For example, `'this is a string'` is a string literal, and `true` is a boolean literal.

Because every variable in Dart refers to an object—an instance of a *class*—you can usually use *constructors* to initialize variables. Some of the built-in types have their own constructors. For example, you can use the `Map()` constructor to create a map, using code such as `new Map()`.

Numbers

Dart numbers come in two flavors:

`int`
: Integers of arbitrary size

`double`
: 64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard

Both `int` and `double` are subtypes of `num`. The num type includes basic operators such as +, -, /, and *, as well as bitwise operators such as >>. The num type is also where you'll find `abs()`, `ceil()`, and `floor()`, among other methods. If num and its subtypes don't have what you're looking for, the Math class might. (In JavaScript produced from Dart code, big integers currently behave differently than they do when the same Dart code runs in the Dart VM.)

Integers are numbers without a decimal point. Here are some examples of defining integer literals:

```
var x = 1;
var hex = 0xDEADBEEF;
var bigInt = 346534658346524376592384765923749587398457294759347029438709349347;
```

If a number includes a decimal, it is a double. Here are some examples of defining double literals:

```
var y = 1.1;
var exponents = 1.42e5;
```

Here's how you turn a string into a number, or vice versa:

```
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

The num type specifies the traditional bitwise shift (<<, >>), AND (&), and OR (|) operators. For example:

```
assert((3 << 1) == 6);  // 0011 << 1 == 0110
assert((3 >> 1) == 1);  // 0011 >> 1 == 0001
assert((3 | 4)  == 7);  // 0011 | 0100 == 0111
```

Strings

A Dart string is a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to just use the other string delimiter.";
```

You can put the value of an expression inside a string by using ${*expression*}. If the expression is an identifier, you can skip the {}. To get the string corresponding to an object, Dart calls the object's `toString()` method.

```
var s = 'string interpolation';
```

```
assert('Dart has $s, which is very handy.' ==
       'Dart has string interpolation, which is very handy.');
assert('That deserves all caps. ${s.toUpperCase()} is very handy!' ==
       'That deserves all caps. STRING INTERPOLATION is very handy!');
```

The == operator tests whether two objects are equivalent. Two strings are equivalent if they have the same characters.

You can concatenate strings using adjacent string literals:

```
var s = 'String ''concatenation'
        " works even over line breaks.";
assert(s == 'String concatenation works even over line breaks.');
```

Another way to create a multi-line string: use a triple quote with either single or double quotation marks.

```
var s1 = '''
You can create
multi-line strings like this one.
''';

var s2 = """This is also a
multi-line string.""";
```

You can create a "raw" string by prefixing it with `r`.

```
var s = r"In a raw string, even \n isn't special.";
```

For more information on using strings, see "Strings and Regular Expressions" (page 54).

Booleans

To represent boolean values, Dart has a type named `bool`. Only two objects have type bool: the boolean literals, `true` and `false`.

When Dart expects a boolean value, only the value `true` is treated as true. All other values are treated as false. Unlike in JavaScript, values such as `1`, `"aString"`, and `someObject` are all treated as false.

For example, consider the following code, which is valid both as JavaScript and as Dart code:

```
var name = 'Bob';
if (name) {
  print('You have a name!'); // Prints in JavaScript, not in Dart.
}
```

If you run this code as JavaScript, without compiling to Dart, it prints "You have a name!" because `name` is a non-null object. However, in Dart running in *production mode*, the above doesn't print at all because `name` is converted to `false` (because `name != true`). In Dart running in *checked mode*, the above code throws an exception because the `name` variable is not a bool.

Here's another example of code that behaves differently in JavaScript and Dart:

```
if (1) {
  print('JavaScript prints this line because it thinks 1 is true.');
} else {
  print('Dart in production mode prints this line.');

  // However, in checked mode, if (1) throws an exception
  // because 1 is not boolean.
}
```

The previous two samples work only in production mode, not checked mode. In checked mode, an exception is thrown if a non-boolean is used when a boolean value is expected.

Dart's treatment of booleans is designed to avoid the strange behaviors that can arise when many values can be treated as true. What this means for you is that, instead of using code like `if (nonbooleanValue)`, you should instead explicitly check for values. For example:

```
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty());

// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Check for null.
var unicorn;
assert(unicorn == null);

// Check for NaN.
var iMeantToDoThis = 0/0;
assert(iMeantToDoThis.isNaN());
```

Lists

Perhaps the most common collection in nearly every programming language is the *array*, or ordered group of objects. In Dart, arrays are List objects, so we usually just call them *lists*.

Dart list literals look like JavaScript array literals. Here's a simple Dart list:

```
var list = [1,2,3];
```

Lists use zero-based indexing, where 0 is the index of the first element and `list.length - 1` is the index of the last element. You can get a list's length and refer to list elements just as you would in JavaScript:

```
var list = [1,2,3];
assert(list.length == 3);
assert(list[1] == 2);
```

The List type and its supertype, Collection, have many handy methods for manipulating lists. For more information about lists, see "Generics" (page 44) and "Collections" (page 57).

Maps

In general, a map is an object that associates keys and values. Dart support for maps is provided by map literals and the Map type.

Here's a simple Dart map:

```
var gifts = {                        // A map literal
// Keys       Values
  'first'  : 'partridge',
  'second' : 'turtledoves',
  'fifth'  : 'golden rings'
};
```

In map literals, each *key* must be a string. If you use a Map constructor, any object can be a key.

```
var map = new Map();                 // use a map constructor.
map[1] = 'partridge';                // key is 1; value is 'partridge'.
map[2] = 'turtledoves';              // key is 2; value is 'turtledoves'.
map[5] = 'golden rings';             // key is 5; value is 'golden rings'.
```

A map *value* can be any object, including null.

You add a new key-value pair to an existing map just as you would in JavaScript:

```
var gifts = { 'first': 'partridge' };
gifts['fourth'] = 'calling birds';    // Add a key-value pair
```

You retrieve a value from a map the same way you would in JavaScript:

```
var gifts = { 'first': 'partridge' };
assert(gifts['first'] == 'partridge');
```

If you look for a key that isn't in a map, you get a null in return.

```
var gifts = { 'first': 'partridge' };
assert(gifts['fifth'] == null);
```

Use `.length` to get the number of key-value pairs in the map:

```
var gifts = { 'first': 'partridge' };
gifts['fourth'] = 'calling birds';
assert(gifts.length == 2);
```

For more information about maps, see "Generics" (page 44) and "Maps" (page 60).

Functions

Here's an example of implementing a function:

```
void printNumber(num number) {
  print('The number is $number.');
}
```

Although the style guide recommends specifying the parameter and return types, you don't have to:

```
printNumber(number) {          // Omitting types is OK.
  print('The number is $number.');
}
```

For functions that contain just one expression, you can use a shorthand syntax:

```
printNumber(number) => print('The number is $number.');
```

The `=> expr;` syntax is a shorthand for `{ return expr;}`. In the `printNumber()` function above, the expression is the call to the top-level `print()` function.

Only an *expression*—not a *statement*—can appear between the arrow (=>) and the semicolon (;). For example, you can't put an if statement (page 28) there, but you can use a conditional (?:) expression (page 28).

You can use types with =>, although the convention is not to do so.

```
printNumber(num number) => print('The number is $number.'); // Types are OK.
```

Here's an example of calling a function:

```
printNumber(123);
```

A function can have two types of parameters: required and optional. The required parameters are listed first, followed by any optional parameters.

Optional Parameters

Optional parameters can be either positional or named, but not both.

Both kinds of optional parameter can have default values. The default values must be compile-time constants such as literals. If no default value is provided, the value is `null`.

If you need to know whether the caller passed in a value for an optional parameter, use the syntax `?param`.

```
if (?device) {    // Returns true if the caller specified the parameter.
  //...The user set the value. Do something with it...
}
```

Optional named parameters

When calling a function, you can specify named parameters using *paramName*: *value*. For example:

```
enableFlags(bold: true, hidden: false);
```

When defining a function, use {*param1*, *param2*, …} to specify named parameters.

```
/// Sets the [bold] and [hidden] flags to the values you specify.
enableFlags({bool bold, bool hidden}) {
  //...
}
```

Use a colon (`:`) to specify default values.

```
/**
 * Sets the [bold] and [hidden] flags to the values you specify,
 * defaulting to false.
 */
enableFlags({bool bold: false, bool hidden: false}) {
  //...
}

enableFlags(bold: true); // bold will be true; hidden will be false.
```

The preceding two examples use documentation comments (page 51).

Optional positional parameters

Wrapping a set of function parameters in `[]` marks them as optional positional parameters.

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Here's an example of calling this function without the optional parameter:

```
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

And here's an example of calling this function with the third parameter:

```
assert(say('Bob', 'Howdy', 'smoke signal') ==
  'Bob says Howdy with a smoke signal');
```

Use = to specify default values.

```
String say(String from, String msg,
  [String device='carrier pigeon', String mood]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  if (mood != null) {
    result = '$result (in a $mood mood)';
  }
  return result;
}

assert(say('Bob', 'Howdy') == 'Bob says Howdy with a carrier pigeon');
```

Functions as First-Class Objects

You can pass a function as a parameter to another function. For example:

```
printElement(element) {
  print(element);
}

var list = [1,2,3];
list.forEach(printElement); // Pass printElement as a parameter.
```

You can also assign a function to a variable, such as:

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') == '!!! HELLO !!!');
```

Lexical Closures

Functions can close over variables defined in surrounding scopes. In the following example, `makeAdder()` captures the variable `n` and makes it available to the function that `makeAdder()` returns. Wherever the returned function goes, it remembers `n`.

```
/// Returns a function that adds [n] to the function's argument.
Function makeAdder(num n) {
  return (num i) => n + i;
}

main() {
  var add2 = makeAdder(2); // Create a function that adds 2.
  var add4 = makeAdder(4); // Create a function that adds 4.
```

```
  assert(add2(3) == 5);
  assert(add4(3) == 7);
}
```

Return Values

All functions return a value. If no return value is specified, the statement `return null;` is implicitly appended to the function body.

Operators

Dart defines the operators shown in Table 2-1. You can override many of these operators, as described in "Operators" (page 41).

Table 2-1. Operators and their precedence

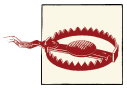
Description	Operator
unary postfix and argument definition test	*expr*++ *expr*-- () [] . ?*identifier*
unary prefix	-*expr* !*expr* ~*expr* ++*expr* --*expr*
multiplicative	* / % ~/
additive	+ -
shift	<< >>
relational and type test	>= > <= < as is is!
equality	== !=
bitwise AND	&
bitwise XOR	^
bitwise OR	|
logical AND	&&
logical OR	||
conditional	*expr1*?*expr2*:*expr3*
cascade	..
assignment	= *= /= ~/= %= += -= <<= >>= &= ^= |=

When you use operators, you create *expressions*. Here are some examples of operator expressions:

```
a++
a + b
a = b
a == b
a? b: c
a is T
```

In Table 2-1, each operator has higher precedence than the operators in the rows below it. For example, the multiplicative operator `%` has higher precedence than (and thus executes before) the equality operator `==`, which has higher precedence than the logical AND operator `&&`. That precedence means that the following two lines of code execute the same way:

```
if ((n % i == 0) && (d % i == 0)) // Parens improve readability.
if (n % i == 0 && d % i == 0)     // Harder to read, but equivalent.
```



For operators that work on two operands, the leftmost operand determines which version of the operator is used. For example, if you have a Vector object and a Point object, `aVector + aPoint` uses the Vector version of +.

Arithmetic Operators

Dart supports the usual arithmetic operators.

Table 2-2. Arithmetic operators

Operator	Meaning
+	Add
–	Subtract
-expr	Unary minus, also known as negation (reverse the sign of the expression)
*	Multiply
/	Divide
~/	Divide, returning an integer result
%	Get the remainder of an integer division (modulo)

Example:

```
var a = 2;
var b = 3;

assert(a + b == 5);
assert(a - b == -1);
assert(a * b == 6);
assert( a / b > 0.6 && a / b < 0.7);
assert(a ~/ b == 0);  // Quotient
assert(a % b == 2);   // Remainder
```

Dart also supports both prefix and postfix increment and decrement operators.

Table 2-3. Increment and decrement operators

Operator	Meaning
++*var*	*var* = *var* + 1 (expression value is *var* + 1)

Operator	Meaning
var++	*var* = *var* + 1 (expression value is *var*)
--*var*	*var* = *var* – 1 (expression value is *var* – 1)
var--	*var* = *var* – 1 (expression value is *var*)

Example:

```
var a, b;

a = 0;
b = ++a;         // Increment a before b gets its value.
assert(a == b);  // 1 == 1

a = 0;
b = a++;         // Increment a AFTER b gets its value.
assert(a != b);  // 1 != 0

a = 0;
b = --a;         // Decrement a before b gets its value.
assert(a == b);  // -1 == -1

a = 0;
b = a--;         // Decrement a AFTER b gets its value.
assert(a != b) ; // -1 != 0
```

Equality and Relational Operators

Table 2-4. Equality and relational operators

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

To test whether two objects *x* and *y* represent the same thing, use the == operator. Here's how the == operator works:

1. If *x* or *y* is null, return true if both are null, and false if only one is null.
2. Return the result of the method invocation *x*.==(*y*). That's right, operators such as == are methods that are invoked on their first operand. You'll see more about this in "Operators" (page 41).

Here's an example of using each of the equality and relational operators:

```
var a = 2;
var b = 3;
var c = a;

assert(a == 2);       // 2 and 2 are equal.
assert(a != b);       // 2 and 3 aren't equal.
assert(b > a);        // 3 is more than 2.
assert(a < b);        // 2 is less then 3.
assert(b >= b);       // 3 is greater than or equal to 3.
assert(a <= b);       // 2 is less than or equal to 3.
```

Type Test Operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Table 2-5. Type test operators

Operator	Meaning
as	Typecast
is	True if the object has the specified type
is!	False if the object has the specified type

The result of `obj is T` is true if `obj` implements the interface specified by `T`. For example, `obj is Object` is always true.

Use the `as` operator to cast an object to a particular type. In general, you should use it as a shorthand for an `is` test on an object following by an expression using that object. For example, consider the following code:

```
if (person is Person) {              // Type check
  person.firstName = 'Bob';
}
```

You can make the code shorter using the `as` operator:

```
(person as Person).firstName = 'Bob';
```

Assignment Operators

As you've already seen, you assign values using the `=` operator. You can also use compound assignment operators such as `+=`, which combine an operation with an assignment.

Table 2-6. Assignment operators

=	*=	%=	&=
+=	/=	<<=	^=
-=	~/=	>>=	|=

Here's how compound assignment operators work:

	Compound assignment	Equivalent expression
For an operator *op*:	`a op= b`	`a = a op b`
Example:	`a += b`	`a = a + b`

The following example uses both assignment and compound assignment operators:

```
var a = 2;          // Assign using =
a *= 3;             // Assign and multiply: a = a * 3
assert(a == 6);
```

Logical Operators

You can invert or combine boolean expressions using the logical operators.

Table 2-7. Logical operators

Operator	Meaning
!*expr*	inverts the following expression (changes false to true, and vice versa)
||	logical OR
&&	logical AND

Here's an example of using the logical operators.

```
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

Bitwise and Shift Operators

You can manipulate the individual bits of numbers in Dart. Usually, you'd use these bitwise and shift operators with integers.

Table 2-8. Bitwise and shift operators

Operator	Meaning
&	AND
|	OR
^	XOR
~*expr*	Unary bitwise complement (0s become 1s; 1s become 0s)
<<	Shift left
>>	Shift right

Here's an example of using bitwise and shift operators.

```
final value = 0x22;
final bitmask = 0x0f;
```

```
assert((value & bitmask)  == 0x02);  // AND
assert((value & ~bitmask) == 0x20);  // AND NOT
assert((value | bitmask)  == 0x2f);  // OR
assert((value ^ bitmask)  == 0x2d);  // XOR
assert((value << 4)       == 0x220); // Shift left
assert((value >> 4)       == 0x02);  // Shift right
```

Other Operators

A few operators remain, most of which you've already seen in other examples.

Table 2-9. Other operators

Operator	Name	Meaning
()	Function application	Represents a function call
[]	List access	Refers to the value at the specified index in the list
expr1`?`*expr2*`:`*expr3*	Conditional	If *expr1* is true, executes *expr2*; otherwise, executes *expr3*
.	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
..	Cascade	Allows you to perform multiple operations on the members of a single object; described in "Classes" (page 34)
`?`*identifier*	Argument definition test	Tests whether the caller specified an optional parameter; described in "Optional Parameters" (page 20)

Control Flow Statements

You can control the flow of your Dart code using any of the following:

- `if` and `else`
- `for` loops
- `while` and `do-while` loops
- `break` and `continue`
- `switch` and `case`
- `assert`

You can also affect the control flow using `try-catch` and `throw`, as explained in "Exceptions" (page 32).

If and Else

Dart supports `if` statements with optional `else` statements. Also see conditional expressions (`?:`), covered in "Other Operators" (page 28).

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

Remember, unlike JavaScript, Dart treats all values other than `true` as `false`. See "Booleans" (page 17) for more information.

For Loops

You can iterate with the standard `for` loop.

```
for (int i = 0; i < candidates.length; i++) {
  candidates[i].interview();
}
```

Closures inside of Dart's `for` loops capture the value of the index, avoiding a common pitfall found in JavaScript. For example, consider:

```
var callbacks = [];
for (var i = 0; i < 2; i++) {
  callbacks.add(() => print(i));
}
callbacks.forEach((c) => c());
```

The output is `0` and then `1`, as expected. In contrast, the example would print `2` and then `2` in JavaScript.

If the object that you are iterating over is a Collection, you can use the `forEach()` method. Using `forEach()` is a good option if you don't need to know the current iteration counter.

```
candidates.forEach((candidate) => candidate.interview());
```

Collections also support the `for-in` form of iteration, as described in "Iteration" (page 64).

```
var collection = [0, 1, 2];
for (var x in collection) {
  print(x);
}
```

While and Do-While

A `while` loop evaluates the condition before the loop.

```
while(!isDone()) {
  doSomething();
}
```

A `do-while` loop evaluates the condition *after* the loop.

```
do {
  printLine();
} while (!atEndOfPage());
```

Break and Continue

Use `break` to stop looping.

```
while (true) {
  if (shutDownRequested()) break;
  processIncomingRequests();
}
```

Use `continue` to skip to the next loop iteration.

```
for (int i = 0; i < candidates.length; i++) {
  var candidate = candidates[i];
  if (candidate.yearsExperience < 5) {
    continue;
  }
  candidate.interview();
}
```

You might write that example differently if you're using a Collection.

```
candidates.filter((c) => c.yearsExperience >= 5)
          .forEach((c) => c.interview());
```

Switch and Case

Switch statements in Dart compare integer, string, or compile-time constants using ==. The compared objects must all be instances of the same class (and not of any of its subtypes), and the class must not override ==.

Each non-empty `case` clause ends with a `break` statement, as a rule. Other valid ways to end a non-empty `case` clause are a `continue`, `throw`, or `return` statement.

Use a `default` clause to execute code when no `case` clause matches.

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
```

```
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
```

The following example omits the `break` statement in the `case` clause, thus generating an error:

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break causes an exception to be thrown!!

  case 'CLOSED':
    executeClosed();
    break;
}
```

However, Dart does support empty `case` clauses, allowing a form of fall-through.

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':     // Empty case falls through.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}
```

If you really want fall-through, you can use a `continue` statement and a label.

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed; // Continues executing at the nowClosed label.

nowClosed:
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}
```

A `case` clause can have local variables, which are visible only inside the scope of that clause.

Assert

Use an `assert` statement to disrupt normal execution if a boolean condition is false. You can find examples of assert statements throughout this tour. Here are some more:

```
assert(text != null);  // Make sure the variable has a non-null value.
assert(number < 100);  // Make sure the value is less than 100.
assert(urlString.startsWith('https')); // Make sure this is an HTTPS URL.
```

Assert statements work only in checked mode. They have no effect in production mode.

Inside the parentheses after `assert`, you can put any expression that resolves to a boolean value or to a function. If the expression's value or function's return value is true, the assertion succeeds and execution continues. Otherwise, the assertion fails and an exception (an AssertionError) is thrown.

Exceptions

Your Dart code can throw and catch exceptions. Exceptions are errors indicating that something unexpected happened. If the exception isn't caught, the isolate that raised the exception is suspended, and typically the isolate and its program are terminated.

In contrast to Java, all of Dart's exceptions are unchecked exceptions. Methods do not declare which exceptions they might throw, and you are not required to catch any exceptions.

Dart provides Exception and Error types, as well as numerous predefined subtypes. You can, of course, define your own exceptions. However, Dart programs can throw any non-null object—not just Exception and Error objects—as an exception.

Throw

Here's an example of throwing, or *raising*, an exception.

```
throw new ExpectException('Value must be greater than zero');
```

You can also throw arbitrary objects.

```
throw 'Out of llamas!';
```

Because throwing an exception is an expression, you can throw exceptions in => statements, as well as anywhere else that allows expressions:

```
String get prettyVersion() => throw const NotImplementedException();
```

Catch

Catching, or capturing, an exception stops the exception from propagating. Catching an exception gives you a chance to handle it.

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  buyMoreLlamas();
}
```

To handle code that can throw more than one type of exception, you can specify multiple catch clauses. The first catch clause that matches the thrown object's type handles the exception. If the catch clause does not specify a type, that clause can handle any type of thrown object.

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {           // A specific exception
  buyMoreLlamas();
} on Exception catch(e) {             // Anything else that is an exception
  print('Unknown exception: $e');
} catch(e) {                          // No specified type, handles all
  print('Something really unknown: $e');
}
```

As the preceding code shows, you can use either `on` or `catch` or both. Use `on` when you need to specify the exception type. Use `catch` when your exception handler needs the exception object.

Finally

To ensure that some code runs whether or not an exception is thrown, use a `finally` clause. If no catch clause matches the exception, the exception is propagated after the finally clause runs.

```
try {
  breedMoreLlamas();
} finally {
  cleanLlamaStalls();  // Always clean up, even if an exception is thrown.
}
```

The finally clause runs after any matching catch clauses.

```
try {
  breedMoreLlamas();
} catch(e) {
  print('Error: $e');  // Handle the exception first.
} finally {
  cleanLlamaStalls();  // Then clean up.
}
```

Learn more by reading "Exceptions" (page 66).

Classes

Dart is an object-oriented language with classes and single inheritance. Every object is an instance of a class, and all classes descend from Object.

To create an object, you can use the `new` keyword with a *constructor* for a class. Constructor names can be either *`ClassName`* or *`ClassName.identifier`*.

```
var jsonData = JSON.parse('{"x":1, "y":2}');

var p1 = new Point(2,2);              // Create a Point using Point().
var p2 = new Point.fromJson(jsonData); // Create a Point using Point.fromJson().
```

Objects have *members* consisting of functions and data (*methods* and *instance variables*, respectively). When you call a method, you *invoke* it on an object: the method has access to that object's functions and data.

Use a dot (.) to refer to an instance variable or method.

```
var p = new Point(2,2);

p.y = 3;            // Set the value of the instance variable y.
assert(p.y == 3);   // Get the value of y.

num distance = p.distanceTo(new Point(4,4)); // Invoke distanceTo() on p.
```

Use the cascade operator (..) when you want to perform a series of operations on the members of a single object.

```
query('#button')
    ..text = 'Click to Confirm'                        // Get an object. Use its
    ..classes.add('important')                         // instance variables
    ..on.click.add((e) => window.alert('Confirmed!')); // and methods.
```

Some classes provide constant constructors. To create a compile-time constant using a constant constructor, use `const` instead of `new`.

```
var p = const ImmutablePoint(2,2);
```

Constructing two identical compile-time constants results in a single, canonical instance.

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a,b)); // They are the same instance!
```

The following sections discuss how to implement classes.

Instance Variables

Here's how you declare instance variables:

```
class Point {
  num x;      // Declare an instance variable (x), initially null.
  num y;      // Declare y, initially null.
  num z = 0;  // Declare z, initially 0.
}
```

All uninitialized instance variables have the value `null`.

All instance variables generate an implicit *getter* method. Non-final, non-const instance variables also generate an implicit *setter* method. For details, see "Getters and setters" (page 39).

```
class Point {
  num x;
  num y;
}

main() {
  var point = new Point();
  point.x = 4;             // Use the setter method for x.
  assert(point.x == 4);    // Use the getter method for x.
  assert(point.y == null); // Values default to null.
}
```

If you initialize an instance variable where it is declared (instead of in a constructor or method), the value is set when the instance is created, which is before the constructor and its initializer list execute.

Constructors

Declare a constructor by creating a function with the same name as its class (plus, optionally, an additional identifier as described in "Named constructors" (page 36)). The most common form of constructor, the generative constructor, creates a new instance of a class.

```
class Point {
  num x;
  num y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

The `this` keyword refers to the current instance.

Use `this` only when there is a name conflict. Otherwise, Dart style omits the `this`.

The pattern of assigning a constructor argument to an instance variable is so common, Dart has syntactic sugar to make it easy.

```
class Point {
  num x;
  num y;

  // Syntactic sugar for setting x and y before the constructor body runs.
  Point(this.x, this.y);
}
```

Default constructors

If you don't declare a constructor, a default constructor is provided for you. The default constructor has no arguments and invokes the no-argument constructor in the superclass.

Constructors aren't inherited

Subclasses don't inherit constructors from their superclass. A subclass that declares no constructors has only the default (no argument, no name) constructor.

Named constructors

Use a named constructor to implement multiple constructors for a class or to provide extra clarity.

```
class Point {
  num x;
  num y;

  Point(this.x, this.y);

  // Named constructor
  Point.fromJson(Map json) {
    x = json['x'];
    y = json['y'];
  }
}
```

Remember that constructors are not inherited, which means that a superclass's named constructor is not inherited by a subclass. If you want a subclass to be created with a named constructor defined in the superclass, you must implement that constructor in the subclass.

Invoking a non-default superclass constructor

By default, a constructor in a subclass calls the superclass's unnamed, no-argument constructor. If the superclass doesn't have such a constructor, then you must manually call one of the constructors in the superclass. Specify the superclass constructor after a colon (:), just before the constructor body (if any).

```
class Person {
  Person.fromJson(Map data) {
    print('in Person');
  }
}

class Employee extends Person {
  // Person does not have a default constructor;
  // you must call super.fromJson(data).
  Employee.fromJson(Map data) : super.fromJson(data) {
    print('in Employee');
  }
}

main() {
  var emp = new Employee.fromJson({});

  // Prints:
  // in Person
  // in Employee
}
```

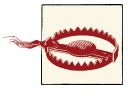
Initializer list

Besides invoking a superclass constructor, you can also initialize instance variables before the constructor body runs. Separate initializers with commas.

```
class Point {
  num x;
  num y;

  Point(this.x, this.y);

  // Initializer list sets instance variables before the constructor body runs.
  Point.fromJson(Map json) : x = json['x'], y = json['y'] {
    print('In Point.fromJson(): ($x, $y)');
  }
}
```



The right-hand side of an initializer does not have access to `this`.

Redirecting constructors

Sometimes a constructor's only purpose is to redirect to another constructor in the same class. A redirecting constructor's body is empty, with the constructor call appearing after a colon (:).

```
class Point {
  num x;
  num y;

  Point(this.x, this.y);               // The main constructor for this class.
  Point.alongXAxis(num x) : this(x, 0); // Delegates to the main constructor.
}
```

Constant constructors

If your class produces objects that never change, you can make these objects compile-time constants. To do this, define a `const` constructor and make sure that all instance variables are `final` or `const`.

```
class ImmutablePoint {
  final num x;
  final num y;
  const ImmutablePoint(this.x, this.y);
  static final ImmutablePoint origin = const ImmutablePoint(0, 0);
}
```

Factory constructors

Use the `factory` keyword when implementing a constructor that doesn't always create a new instance of its class. For example, a factory constructor might return an instance from a cache, or it might return an instance of a subtype.

The following example demonstrates a factory constructor returning objects from a cache.

```
class Logger {
  final String name;
  bool mute = false;

  // _cache is library-private, thanks to the _ in front of its name.
  static final Map<String, Logger> _cache = <String, Logger>{};

  factory Logger(String name) {
    if (_cache.containsKey(name)) {
      return _cache[name];
    } else {
      final logger = new Logger._internal(name);
      _cache[name] = logger;
      return logger;
    }
  }
```

```
    Logger._internal(this.name);

    void log(String msg) {
      if (!mute) {
        print(msg);
      }
    }
}
```

Factory constructors have no access to `this`.

To invoke a factory constructor, you use the `new` keyword:

```
var logger = new Logger('UI');
logger.log('Button clicked');
```

Methods

Methods are functions that provide behavior for an object.

Instance methods

Instance methods on objects can access instance variables and `this`. The `distanceTo()` method in the following sample is an example of an instance method.

```
class Point {
  num x;
  num y;
  Point(this.x, this.y);

  num distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

Getters and setters

Getters and setters are special methods that provide read and write access to an object's properties. Recall that each instance variable has an implicit getter, plus a setter if appropriate. You can create additional properties by implementing getters and setters, using the `get` and `set` keywords.

```
class Rectangle {
  num left;
  num top;
  num width;
```

```
  num height;

  Rectangle(this.left, this.top, this.width, this.height);

  // Define two calculated properties: right and bottom.
  num get right             => left + width;
      set right(num value)  => left = value - width;
  num get bottom            => top + height;
      set bottom(num value) => top = value - height;
}

main() {
  var rect = new Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

With getters and setters, you can start with instance variables, later wrapping them with methods, all without changing client code.

Operators such as increment (++) work in the expected way, whether or not a getter is explicitly defined. To avoid any unexpected side effects, the operator calls the getter exactly once, saving its value in a temporary variable.

Abstract methods

Instance, getter, and setter methods can be abstract, defining an interface but leaving its implementation up to other classes. To make a method abstract, use a semicolon (;) instead of a method body.

```
abstract class Doer {
  // ...Define instance variables and methods...

  void doSomething(); // Define an abstract method.
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // ...Provide an implementation, so the method is not abstract here...
  }
}
```

Calling an abstract method results in a run-time error.

Also see "Abstract Classes" (page 41).

Operators

You can override the operators shown in Table 2-10. For example, if you define a Vector class, you might define a + method to add two vectors.

Table 2-10. Operators that can be overridden

<	+	|	[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
–	%	>>	

Here's an example of a class that overrides the + and - operators.

```
class Vector {
  final int x;
  final int y;
  const Vector(this.x, this.y);

  Vector operator +(Vector v) { // Overrides + (a + b).
    return new Vector(x + v.x, y + v.y);
  }

  Vector operator -(Vector v) { // Overrides - (a - b).
    return new Vector(x - v.x, y - v.y);
  }
}

main() {
  final v = new Vector(2,3);
  final w = new Vector(2,2);

  assert(v.x == 2 && v.y == 3);         // v   == (2,3)
  assert((v+w).x == 4 && (v+w).y == 5); // v+w == (4,5)
  assert((v-w).x == 0 && (v-w).y == 1); // v-w == (0,1)
}
```

For an example of overriding ==, see "Implementing map keys" (page 63).

Abstract Classes

Use the `abstract` modifier to define an *abstract class*—a class that can't be instantiated. Abstract classes are useful for defining interfaces, often with some implementation. If you want your abstract class to appear to be instantiable, define a factory constructor (page 38).

Abstract classes often have abstract methods (page 40). Here's an example of declaring an abstract class that has an abstract method.

```
// This class is declared abstract and thus can't be instantiated.
abstract class AbstractContainer {
  // ...Define constructors, fields, methods...

  void updateChildren(); // Abstract method.
}
```

The following class isn't abstract, and thus can be instantiated even though it defines an abstract method.

```
class SpecializedContainer extends AbstractContainer {
  // ...Define more constructors, fields, methods...

  void updateChildren() {
    // ...Implement updateChildren()...
  }
// Abstract method causes a warning but doesn't prevent instantiatation.
  void doSomething();
}
```

Implicit Interfaces

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface.

A class implements one or more interfaces by declaring them in an `implements` clause and then providing the APIs required by the interfaces. For example:

```
// A person. The implicit interface contains greet().
class Person {
  final _name;          // In the interface, but visible only in this library,
  Person(this._name);   // Not in the interface, since this is a constructor.
  String greet(who) => 'Hello, $who. I am $_name.'; // In the interface.
}

// An implementation of the Person interface.
class Imposter implements Person {
  final _name = "";      // We have to define this, but we don't use it.
  String greet(who) => 'Hi $who. Do you know who I am?';
}

greetBob(Person person) => person.greet('bob');

main() {
  print(greetBob(new Person('kathy')));
  print(greetBob(new Imposter()));
}
```

Here's an example of specifying that a class implements multiple interfaces:

```
class Point implements Comparable, Location {
  //...
}
```

Extending a Class

Use `extends` to create a subclass, and `super` to refer to the superclass.

```
class Television {
  void turnOn() {
    _illuminateDisplay();
    _activateIrSensor();
  }
}

class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    _bootNetworkInterface();
    _initializeMemory();
    _upgradeApps();
  }
}
```

Subclasses can override instance methods, getters, and setters.

Class Variables and Methods

Use the `static` keyword to implement class-wide variables and methods.

Static variables

Static variables (class variables) are useful for class-wide state and constants.

```
class Color {
  static const RED = const Color('red'); // A constant static variable.
  final String name;                     // An instance variable.
  const Color(this.name);                // A constant constructor.
}

main() {
  assert(Color.RED.name == 'red');
}
```

Static variables aren't initialized until they're used.

Static methods

Static methods (class methods) do not operate on an instance, and thus do not have access to `this`.

```
class Point {
  num x;
```

```
  num y;
  Point(this.x, this.y);

  static num distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

main() {
  var a = new Point(2, 2);
  var b = new Point(4, 4);
  var distance = Point.distanceBetween(a,b);
  assert(distance < 2.9 && distance > 2.8);
}
```

Consider using top-level functions, instead of static methods, for common or widely used utilities and functionality.

You can use static methods as compile-time constants. For example, you can pass a static method as a parameter to a constant constructor.

Generics

If you look at the API documentation for the basic array type, List, you'll see that the type is actually `List<E>`. The <...> notation marks List as a *generic* (or *parameterized*) type—a type that has formal type parameters. By convention, type variables have single-letter names, such as E, T, S, K, and V.

Why Use Generics?

Because types are optional in Dart, you never *have* to use generics. You might *want* to, though, for the same reason you might want to use other types in your code: types (generic or not) let you document and annotate your code, making your intent clearer.

For example, if you intend for a list to contain only strings, you can declare it as `List<String>` (read that as "list of string"). That way you, your fellow programmers, and your tools (such as Dart Editor and the Dart VM in checked mode) can detect that assigning a non-string to the list is probably a mistake.

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
//...
names.add(42); // Fails in checked mode (succeeds in production mode).
```

Another reason for using generics is to reduce code duplication. Generics let you share a single interface and implementation between many types, while still taking advantage of checked mode and static analysis early warnings. For example, say you create an interface for caching an object:

```
abstract class ObjectCache {
  Object getByKey(String key);
  setByKey(String key, Object value);
}
```

You discover that you want a string-specific version of this interface, so you create another interface:

```
abstract class StringCache {
  String getByKey(String key);
  setByKey(String key, String value);
}
```

Later, you decide you want a number-specific version of this interface... You get the idea.

Generic types can save you the trouble of creating all these interfaces. Instead, you can create a single interface that takes a type parameter:

```
abstract class Cache<T> {
  T getByKey(String key);
  setByKey(String key, T value);
}
```

In this code, T is the stand-in type. It's a placeholder that you can think of as a type that a developer will define later.

Using Collection Literals

List and map literals can be parameterized. Parameterized literals are just like the literals you've already seen, except that you add `<type>` (for lists) or `<keyType, valueType>` (for maps) before the opening bracket. You might use parameterized literals when you want type warnings in checked mode. Here is example of using typed literals:

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var pages = <String, String>{
    'index.html':'Homepage',
    'robots.txt':'Hints for web robots',
    'humans.txt':'We are people, not machines' };
```

Map literals always have string *keys*, so their type is always `<String, SomeType>`.

Using Constructors

To specify one or more types when using a constructor, put the types in angle brackets (`<...>`) just after the class name. For example:

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
var nameSet = new Set<String>.from(names);
```

The following code creates a map that has integer keys and values of type View:

```
var views = new Map<int, View>();
```

Generic Collections and the Types they Contain

Dart generic types are *reified*, which means that they carry their type information around at runtime. For example, you can test the type of a collection, even in production mode:

```
var names = new List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

However, the `is` expression checks the type of the *collection* only—not of the objects inside it. In production mode, a List<String> might have some non-string items in it. The solution is to either check each item's type or wrap item-manipulation code in an exception handler (see "Exceptions" (page 32)).

In contrast, generics in Java use *erasure*, which means that generic type parameters are removed at runtime. In Java, you can test whether an object is a List, but you can't test whether it's a List<String>.

For more information about generics, see Optional Types in Dart.

Libraries and Visibility

The `import`, `part`, and `library` directives can help you create a modular and shareable code base. Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (_) are visible only inside the library. *Every Dart app is a library*, even if it doesn't use a library directive.

Libraries can be distributed using packages. See "pub: The Dart Package Manager" (page 91) for information about pub, a package manager included in the SDK.

Using Libraries

Use `import` to specify how a namespace from one library is used in the scope of another library.

For example, Dart web apps generally use the dart:html library, which they can import like this:

```
import 'dart:html';
```

The only required argument to `import` is a URI[1] specifying the library. For built-in libraries, the URI has the special `dart:` scheme. For other libraries, you can use a file system path or the `package:` scheme. The `package:` scheme specifies libraries provided by a package manager such as the pub tool. For example:

```
import 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

Specifying a library prefix

If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
//...
var element1 = new Element();      // Uses Element from lib1.
var element2 = new lib2.Element(); // Uses Element from lib2.
```

Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
import 'package:lib1/lib1.dart' show foo, bar; // Import only foo and bar.
import 'package:lib2/lib2.dart' hide foo;      // Import all names EXCEPT foo.
```

Implementing Libraries

Use the `part` directive to specify the files that are part of a library, and `library` to specify that a file declares a library.

Using multiple files

Whether or not you specify a `library` directive, you can use `part` to specify the files that implement the current library.

```
// No library directive; this file defines an anonymous library.
part 'ball.dart';   // Part of this library's implementation is in ball.dart.
part 'util.dart';   // Another part is in util.dart.

import 'dart:html'; // This app uses the HTML library.
```

1. URI stands for *uniform resource identifier*. URLs (*uniform resource locators*) are a common kind of URI.

```
//...
main() {    // Having a main() method makes this an app (and thus a library).
  //...
}
```

Declaring a library

To explicitly declare a library, use a `library` statement. For example:

```
library ballgame;   // Declare that this is a library.

part 'ball.dart';   // Part of this library's implementation is in ball.dart.
part 'util.dart';   // Another part is in util.dart.

import 'dart:html'; // This app uses the HTML library.
//...
main() {            // We could move this code to another file in this library.
  //...
}
```

Associating a file with a library

You can use `part` followed by `of` (`part of`) in implementation files to specify that the file is associated with a library. Using `part of` is optional, but it helps tools such as Dart Editor. Here's an example of using `part` and `part of`:

```
// In ballgame.dart:
library ballgame;

import 'dart:utf';
// ...Other imports go here...

part 'ball.dart';
part 'util.dart';

//...


// In ball.dart:
part of ballgame;

// ...code goes here...


// In util.dart:
part of ballgame;

// ...code goes here...
```

Re-exporting libraries

You can combine or repackage libraries by re-exporting part or all of them. For example, you might have a huge library that you implement as a set of smaller libraries. Or you might create a library that provides a subset of methods from another library.

```
// In french.dart:
library french;
hello() => print('Bonjour!');
goodbye() => print('Au Revoir!');

// In togo.dart:
library togo;
import 'french.dart';
export 'french.dart' show hello;

// In another .dart file:
import 'togo.dart';

void main() {
  hello();   //print bonjour
  goodbye(); //FAIL
}
```

Isolates

Modern web browsers, even on mobile platforms, run on multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code.

Instead of threads, all Dart code runs inside of *isolates*. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate.

Learn more about isolates in "dart:isolate - Concurrency with Isolates" (page 76).

Typedefs

In Dart, functions are objects, just like strings and numbers are objects. A *typedef*, or *function-type alias*, gives a function type a name that you can use when declaring fields and return types. A typedef retains type information when a function type is assigned to a variable.

Consider the following code, which does not use a typedef.

```
class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    compare = f;
```

```
  }
}

int sort(Object a, Object b) => ... ; // Initial, broken implementation.

main() {
  SortedCollection collection = new SortedCollection(sort);

  // All we know is that compare is a function, but what type of function?
  assert(collection.compare is Function);
}
```

Type information is lost when assigning `f` to `compare`. The type of `f` is `(Object, Object)` → `int` (where → means returns), yet the type of `compare` is Function. If we change the code to use explicit names and retain type information, both developers and tools can use that information.

```
typedef int Compare(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

int sort(Object a, Object b) => ... ; // Initial, broken implementation.

main() {
  SortedCollection collection = new SortedCollection(sort);
  assert(collection.compare is Function);
  assert(collection.compare is Compare);
}
```

Currently, typedefs are restricted to function types. We expect this to change.

Because typedefs are simply aliases, they offer a way to check the type of any function. For example:

```
typedef int Compare(int a, int b);

int sort(int a, int b) => a - b;

main() {
  assert(sort is Compare);  // True!
}
```

Comments

Dart supports single-line comments, multi-line comments, and documentation comments.

Single-Line Comments

A single-line comment begins with `//`. Everything between `//` and the end of line is ignored by the Dart compiler.

```
main() {
  // TODO: refactor into an AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}
```

Multi-Line Comments

A multi-line comment begins with `/*` and ends with `*/`. Everything between `/*` and `*/` is ignored by the Dart compiler (unless the comment is a documentation comment; see the next section). Multi-line comments can nest.

```
main() {
  /*
   * This is a lot of work. Consider raising chickens.

  Llama larry = new Llama();
  larry.feed();
  larry.exercise();
  larry.clean();
   */
}
```

Documentation Comments

Documentation comments are multi-line or single-line comments that begin with `/**` or `///`. Using `///` on consecutive lines has the same effect as a multi-line doc comment.

Inside a documentation comment, the Dart compiler ignores all text unless it is enclosed in brackets. Using brackets, you can refer to classes, methods, fields, top-level variables, functions, and parameters. The names in brackets are resolved in the lexical scope of the documented program element.

Here is an example of documentation comments with references to other classes and arguments:

```
/**
 * The llama (Lama glama) is a domesticated South American
 * camelid, widely used as a meat and pack animal by Andean
 * cultures since pre-Hispanic times.
```

```
 */
class Llama {
  String name;

  /**
   * Feeds your llama [Food].
   *
   * The typical llama eats one bale of hay per week.
   */
  void feed(Food food) {
    //...
  }

  /// Exercises your llama with an [activity] for
  /// [timeLimit] minutes.
  void exercise(Activity activity, int timeLimit) {
    //...
  }
}
```

In the generated documentation, `[Food]` becomes a link to the API docs for the Food class.

To parse Dart code and generate HTML documentation, you can use Dart Editor, which in turn uses the SDK's dartdoc package. For an example of generated documentation, see the Dart API documentation.

Summary

This chapter summarized the commonly used features in the Dart language. For more information about the language, see the Dart Language Specification and articles such as Idiomatic Dart.

CHAPTER 3

A Tour of the Dart Libraries

This chapter shows you how to use the major features in Dart's libraries. It's just an overview, and by no means comprehensive. Whenever you need more details about a class, consult the Dart API reference. Expect major changes to the Dart libraries before Dart's first production release.

dart:core - Numbers, Collections, Strings, and More

The Dart core library provides a small but critical set of built-in functionality. This library is automatically imported into every Dart program.

Numbers

The dart:core library defines the num, int, and double classes, which have some basic utilities for working with numbers.

You can convert a string into an integer or double with the `parse()` methods of int and double, respectively.

```
assert(int.parse('42') == 42);
assert(double.parse('0.50') == 0.5);
```

Use the toString() method (defined by Object) to convert an int or double to a string. To specify the number of digits to the right of the decimal, use `toStringAsFixed()` (defined by num). To specify the number of significant digits in the string, use `toStringAsPrecision()` (also in num).

```
// Convert an int to a string.
assert(42.toString() == '42');

// Convert a double to a string.
assert(123.456.toString() == '123.456');
```

```
// Specify the number of digits after the decimal.
assert(123.456.toStringAsFixed(2) == '123.46');

// Specify the number of significant figures.
assert(123.456.toStringAsPrecision(2) == '1.2e+2');
assert(double.parse('1.2e+2') == 120.0);
```

For more information, see the API documentation for int, double, and num. Also see "dart:math - Math and Random" (page 66).

Strings and Regular Expressions

A string in Dart is an immutable sequence of UTF-16 code units. The language tour has more information about strings (page 16). You can use regular expressions (RegExp objects) to search within strings and to replace parts of strings.

The String class defines such methods as `split()`, `contains()`, `startsWith()`, `endsWith()`, and more.

Searching inside a string

You can find particular locations within a string, as well as check whether a string begins with or ends with a particular pattern.

```
// Check whether a string contains another string.
assert('Never odd or even'.contains('odd'));

// Does a string start with another string?
assert('Never odd or even'.startsWith('Never'));

// Does a string end with another string?
assert('Never odd or even'.endsWith('even'));

// Find the location of a string inside a string.
assert('Never odd or even'.indexOf('odd') == 6);
```

Extracting data from a string

You can get the individual characters or character codes from a string as Strings or ints, respectively.

You can also extract a substring or split a string into a list of substrings.

```
// Grab a substring.
assert('Never odd or even'.substring(6, 9) == 'odd');

// Split a string using a string pattern.
var parts = 'structured web apps'.split(' ');
assert(parts.length == 3);
assert(parts[0] == 'structured');

// Get the character (as a string) by index.
```

```
assert('Never odd or even'[0] == 'N');

// Use splitChars() to get a list of all characters (as Strings);
// good for iterating.
for (var char in 'hello'.splitChars()) {
  print(char);
}

// Get the char code at an index.
assert('Never odd or even'.charCodeAt(0) == 78);

// Get all the char codes as a list of integers.
var charCodes = 'Never odd or even'.charCodes();
assert(charCodes.length == 17);
assert(charCodes[0] == 78);
```

Converting to uppercase or lowercase

You can easily convert strings to their uppercase and lowercase variants.

```
// Convert to uppercase.
assert('structured web apps'.toUpperCase() == 'STRUCTURED WEB APPS');

// Convert to lowercase.
assert('STRUCTURED WEB APPS'.toLowerCase() == 'structured web apps');
```

Trimming and empty strings

Remove all leading and trailing white space with `trim()`. To check whether a string is empty (length is zero), use `isEmpty()`.

```
// Trim a string.
assert('  hello  '.trim() == 'hello');

// Check whether a string is empty.
assert(''.isEmpty());

// Strings with only white space are not empty.
assert(!'  '.isEmpty());
```

Replacing part of a string

Strings are immutable objects, which means you can create them but you can't change them. If you look closely at the String API docs, you'll notice that none of the methods actually changes the state of a String. For example, the method `replaceAll()` returns a new String without changing the original String.

```
var greetingTemplate = 'Hello, NAME!';
var greeting = greetingTemplate.replaceAll(new RegExp('NAME'), 'Bob');

assert(greeting != greetingTemplate); // greetingTemplate didn't change.
```

Building a string

To programmatically generate a string, you can use StringBuffer. A StringBuffer doesn't generate a new String object until `toString()` is called.

```
var sb = new StringBuffer();

sb.add('Use a StringBuffer ');
sb.addAll(['for ', 'efficient ', 'string ', 'creation ']);
sb.add('if you are ').add('building lots of strings.');

var fullString = sb.toString();

assert(fullString ==
    'Use a StringBuffer for efficient string creation '
    'if you are building lots of strings.');

sb.clear();  // All gone!
assert(sb.toString() == '');
```

Regular expressions

The RegExp class provides the same capabilities as JavaScript regular expressions. Use regular expressions for efficient searching and pattern matching of strings.

```
// A regular expression for one or more digits
var numbers = const RegExp(r'\d+');

var allCharacters = 'llamas live fifteen to twenty years';
var someDigits = 'llamas live 15 to 20 years';

// Contains() can use a regular expression.
assert(!allCharacters.contains(numbers));
assert(someDigits.contains(numbers));

// Replace every match with another string.
var exedOut = someDigits.replaceAll(numbers, 'XX');
assert(exedOut == 'llamas live XX to XX years');
```

You can work directly with the RegExp class, too. The Match class provides access to a regular expression match.

```
var numbers = const RegExp(r'\d+');
var someDigits = 'llamas live 15 to 20 years';

// Check whether the reg exp has a match in a string.
assert(numbers.hasMatch(someDigits));

// Loop through all matches.
for (var match in numbers.allMatches(someDigits)) {
  print(match.group(0)); // 15, then 20
}
```

More information

Refer to the String API docs for a full list of methods. Also see the API docs for StringBuffer, Pattern, RegExp, and Match.

Collections

Dart ships with a core collections API, which includes classes for lists, sets, and maps.

Lists

As the language tour shows, you can use literals to create and initialize lists (page 18). Alternatively, use one of the List constructors. The List class also defines several methods for adding items to and removing items from lists.

```
// Use a List constructor.
var vegetables = new List();

// Or simply use a list literal.
var fruits = ['apples', 'oranges'];

// Add to a list.
fruits.add('kiwis');

// Add multiple items to a list.
fruits.addAll(['grapes', 'bananas']);

// Get the list length.
assert(fruits.length == 5);

// Remove a single item.
var appleIndex = fruits.indexOf('apples');
fruits.removeAt(appleIndex);
assert(fruits.length == 4);

// Remove all elements from a list.
fruits.clear();
assert(fruits.length == 0);
```

Use `indexOf()` to find the index of an object in a list.

```
var fruits = ['apples', 'oranges'];

// Access a list item by index.
assert(fruits[0] == 'apples');

// Find an item in a list.
assert(fruits.indexOf('apples') == 0);
```

Sort a list using the `sort()` method. You must provide a sorting function that compares two objects. This sorting function must return < 0 for *smaller*, 0 for the *same*, and > 0 for *bigger*. The following example uses `compareTo()`, which is defined by Comparable and implemented by String.

```
var fruits = ['bananas', 'apples', 'oranges'];

// Sort a list.
fruits.sort((a, b) => a.compareTo(b));
assert(fruits[0] == 'apples');
```

Lists are parameterized types, so you can specify the type that a list should contain.

```
// This list should contain only strings.
var fruits = new List<String>();

fruits.add('apples');
var fruit = fruits[0];
assert(fruit is String);

// Generates static analysis warning, num is not a string.
fruits.add(5);  // BAD: Throws exception in checked mode.
```

Refer to the List API docs for a full list of methods.

Sets

A set in Dart is an unordered collection of unique items. Because a set is unordered, you can't get a set's items by index (position).

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
assert(ingredients.length == 3);

// Adding a duplicate item has no effect.
ingredients.add('gold');
assert(ingredients.length == 3);

// Remove an item from a set.
ingredients.remove('gold');
assert(ingredients.length == 2);
```

Use `contains()` and `containsAll()` to check whether one or more objects are in a set.

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);

// Check whether an item is in the set.
assert(ingredients.contains('titanium'));

// Check whether all the items are in the set.
assert(ingredients.containsAll(['titanium', 'xenon']));
```

An intersection is a set whose items are in two other sets. A subset has all of its items included in another, potentially larger, collection.

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);

// Create the intersection of two sets.
var nobleGases = new Set.from(['xenon', 'argon']);
var intersection = ingredients.intersection(nobleGases);
assert(intersection.length == 1);
assert(intersection.contains('xenon'));

// Check whether this set is a subset of another collection.
// That is, does another collection contains all the items of this set?
var allElements = ['hydrogen', 'helium', 'lithium', 'beryllium',
                   'gold', 'titanium', 'xenon' /* all the rest */];
assert(ingredients.isSubsetOf(allElements));
```

Refer to the Set API docs for a full list of methods.

Common collection methods

Both List and Set extend the Collection class. As such, they share common functionality found in all collections. The following examples work with any object that implements Collection.

Use `isEmpty()` to check whether a collection has no items.

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];
assert(!teas.isEmpty());
```

To apply a function to each item in a collection, you can use `forEach()`. Or use `map()` if you want a new collection that contains the results.

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

teas.forEach((tea) => print('I drink $tea'));

var loudTeas = teas.map((tea) => tea.toUpperCase());
assert(loudTeas.some((tea) => tea == 'GREEN'));
```

Use `some()` and `every()` to check whether some or all items in a collection match a condition.

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

// Chamomile is not caffeinated.
isDecaffeinated(String teaName) => teaName == 'chamomile';

// Use filter() to create a new collection with only the items
// that return true from the provided function.
var decaffeinatedTeas = teas.filter((tea) => isDecaffeinated(tea));
// or teas.filter(isDecaffeinated)
```

```
// Use some() to check whether at least one item in the collection
// satisfies a condition.
assert(teas.some(isDecaffeinated));

// Use every() to check whether all the items in a collection
// satisfy a condition.
assert(!teas.every(isDecaffeinated));
```

Refer to the Collection API docs for a full list of methods.

Maps

A map, commonly known as a *dictionary* or *hash*, is an unordered collection of key-value pairs. Maps associate a key to some value for easy retrieval. Unlike in JavaScript, Dart objects are not maps.

The Map class does not itself extend Collection. You can, however, get a collection of a map's keys or its values.

You can declare a map using a terse literal syntax, or you can use a traditional constructor.

```
// Map literals use strings as keys.
var hawaiianBeaches = {
  'oahu' : ['waikiki', 'kailua', 'waimanalo'],
  'big island' : ['wailea bay', 'pololu beach'],
  'kauai' : ['hanalei', 'poipu']
};

// Maps can be built from a constructor.
var searchTerms = new Map();

// Maps are parameterized types; you can specify what types
// the key and value should be.
var nobleGases = new Map<int, String>();
```

You add, get, and set map items using the bracket syntax. Use `remove()` to remove a key and its value from a map.

```
var nobleGases = new Map<int, String>();

// Maps from constructors can use any object as a key.
// Integers and strings are common key types.
nobleGases[54] = 'xenon';

// Retrieve a value with a key.
assert(nobleGases[54] == 'xenon');

// Check whether a map contains a key.
assert(nobleGases.containsKey(54));
```

```
// Remove a key and its value.
nobleGases.remove(54);
assert(!nobleGases.containsKey(54));
```

You can retrieve all the values or all the keys from a map.

```
var hawaiianBeaches = {
  'oahu' : ['waikiki', 'kailua', 'waimanalo'],
  'big island' : ['wailea bay', 'pololu beach'],
  'kauai' : ['hanalei', 'poipu']
};

// Get all the keys as an unordered collection (a list).
var keys = hawaiianBeaches.getKeys();

assert(keys.length == 3);
assert(new Set.from(keys).contains('oahu'));

// Get all the values as an unordered collection (a list of lists).
var values = hawaiianBeaches.getValues();
assert(values.length == 3);
assert(values.some((v) => v.indexOf('waikiki') != -1));
```

You can also iterate through the key-value pairs.

```
// NOTE: Do not depend on iteration order.
hawaiianBeaches.forEach((k,v) {
  print('I want to visit $k and swim at $v');
  // I want to visit oahu and swim at [waikiki, kailua, waimanalo], etc.
});
```

To check whether a map contains a key, use `containsKey()`. Because map values can be null, you cannot rely on simply getting the value for the key and checking for null to determine the existence of a key.

```
var hawaiianBeaches = {
  'oahu' : ['waikiki', 'kailua', 'waimanalo'],
  'big island' : ['wailea bay', 'pololu beach'],
  'kauai' : ['hanalei', 'poipu']
};

assert(hawaiianBeaches.containsKey('oahu'));
assert(!hawaiianBeaches.containsKey('florida'));
```

Use the `putIfAbsent()` method when you want to assign a value to a key if and only if the key does not already exist in a map. You must provide a function that returns the value.

```
var teamAssignments = {};
teamAssignments.putIfAbsent('Catcher', () => pickToughestKid());
assert(teamAssignments['Catcher'] != null);
```

Refer to the Map API docs for a full list of methods.

Dates and Times

A Date object is a point in time. The time zone is either UTC or the local time zone.

You can create Date objects using several constructors.

```
// Get the current date and time.
var now = new Date.now();

// Create a new Date with the local time zone.
var y2k = new Date(2000, 1, 1, 0, 0, 0, 0);

// You can also use named parameters.
y2k = new Date(2000, month: 1, day: 1, hour: 0, minute: 0, second: 0,
               millisecond: 0);

// Specify all the parts of a date as a UTC time.
y2k = new Date(2000, 1, 1, 0, 0, 0, 0, isUtc: true);

// Specify a UTC date and time in milliseconds since the Unix epoch.
y2k = new Date.fromMillisecondsSinceEpoch(946684800000, isUtc: true);

// Parse an ISO 8601 date.
y2k = new Date.fromString('2000-01-01T00:00:00Z');
```

The `millisecondsSinceEpoch` property of a date returns the number of milliseconds since the epoch.

```
var y2k = new Date.fromString('2000-01-01T00:00:00Z');
assert(y2k.millisecondsSinceEpoch == 946684800000);
```

Use the Duration class to calculate the difference between two dates and to shift a date's time forward or backwards.

```
var y2k = new Date.fromString('2000-01-01T00:00:00Z');

// Add one year.
var y2001 = y2k.add(const Duration(days: 366));
assert(y2001.year == 2001);

// Subtract 30 days.
var december2000 = y2001.subtract(const Duration(days: 30));
assert(december2000.year == 2000);
assert(december2000.month == 12);

// Calculate the difference between two dates.
// Returns a Duration object.
var duration = y2001.difference(y2k);
assert(duration.inDays == 366); // y2k was a leap year.
```

Refer to the API docs for Date and Duration for a full list of methods.

Utility Classes

The core library contains various utility classes, useful for sorting, mapping values, and iterating.

Comparing objects

Implement the Comparable interface to indicate that an object can be compared to another object, usually for sorting. The `compareTo()` method returns < 0 for *smaller*, 0 for the *same*, and > 0 for *bigger*.

```
class Line implements Comparable {
  final length;
  const Line(this.length);
  int compareTo(Line other) => length - other.length;
}

main() {
  var short = const Line(1);
  var long = const Line(100);
  assert(short.compareTo(long) < 0);
}
```

Implementing map keys

Each object in Dart automatically provides an integer hash code, and thus can be used as a key in a map. However, you can override the `hashCode()` method to generate a custom hash code. If you do, be sure to override the `==` operator, as well. Objects that are equal (via `==`) must have identical hash codes. A hash code doesn't have to be unique, but it should be well distributed.

```
class Person {
  String firstName, lastName;

  Person(this.firstName, this.lastName);

  // Override hashCode using strategy from Effective Java, Chapter 11.
  int hashCode() {
    int result = 17;
    result = 37 * result + firstName.hashCode();
    result = 37 * result + lastName.hashCode();
    return result;
  }

  // Always implement operator== if you override hashCode.
  bool operator==(other) {
    if (identical(other, this)) return true;
    return (other.firstName == firstName && other.lastName == lastName);
  }
}
```

```
main() {
  var p1 = new Person('bob', 'smith');
  var p2 = new Person('bob', 'smith');
  assert(p1.hashCode() == p2.hashCode());
}
```

Iteration

The Iterable and Iterator classes support for-in loops. Implement Iterable to signal that an object can provide an Iterator, and thus be used by for-in loops. Implement Iterator to define the actual iteration ability.

```
class Process {
  // Represents a process...
}

class ProcessIterator implements Iterator<Process> {
  Process next() {
    // Return the next process if possible; but if not:
    throw new NoMoreElementsException();
  }
  bool hasNext() {
    // True if calling next() would return a process
    return false;
  }
}

// A mythical class that lets you iterate through all processes.
class Processes implements Iterable<Process> {
  Iterator<Process> iterator() {
    return new ProcessIterator();
  }
}

main() {
  // Objects that implement Iterable can be used with for-in.
  for (var process in new Processes()) {
    // Do something with the process.
  }
}
```

Asynchronous Programming

Asynchronous programming often uses callback functions, but Dart provides an alternative: Future objects. A Future is like a promise for a result to be provided sometime in the future.

You have the option of using a Completer to produce a Future and, later, to supply a value to the Future.

```
Future<bool> longExpensiveSearch() {
  var completer = new Completer();
```

```
  // Perform exhaustive search.
  // ...
  // Sometime later,
  // found it!!
  completer.complete(true);
  return completer.future;
}

main() {
  var result = longExpensiveSearch(); // Returns immediately.

  // result.then() returns immediately.
  result.then((success) {
    // The following code executes when the operation is complete.
    print('The item was found: $success');
  });
}
```

Chaining multiple asynchronous methods

The Future class specifies a `chain()` method, which is a useful way to specify that multiple asynchronous methods run in a certain order.

```
Future result = costlyQuery();
result.handleException((exception) => print('DOH!'));

result.chain((value) => expensiveWork())
      .chain((value) => lengthyComputation())
      .then((value)  => print('done!'));
```

In the above example, the methods run in the following order:

1. `costlyQuery()`
2. `expensiveWork()`
3. `lengthyComputation()`

Waiting for multiple futures

Sometimes your algorithm needs to initiate many asynchronous methods and wait for each one to complete before continuing. Use the Futures class to manage multiple Futures and wait for them all to complete.

```
Future deleteDone = deleteLotsOfFiles();
Future copyDone = copyLotsOfFiles();
Future checksumDone = checksumLotsOfOtherFiles();

Futures.wait([deleteDone, copyDone, checksumDone]).then((List values) {
  print('Done with all the long steps');
});
```

More information

For examples of using Future, see "dart:io - I/O for Command-Line Apps" (page 80).

Exceptions

The Dart core library defines many common exceptions and errors. Exceptions are considered conditions that you can plan ahead for and catch. Errors are conditions that you don't expect or plan for.

Some of the most common exceptions and errors include:

NoSuchMethodError
: Thrown when a receiving object does not implement a method.

NullPointerException
: Thrown when the program tries to call a method or access a field of a null object.

ArgumentError
: Can be thrown by a method that encounters an unexpected argument.

Throwing an application-specific exception is a common way to indicate that an error has occurred. You can define a custom exception by implementing the Exception interface.

```
class FooException implements Exception {
  final String msg;
  const FooException([this.msg]);
  String toString() => msg == null ? 'FooException' : msg;
}
```

For more information, see "Exceptions" (page 32) and the Exception API docs.

dart:math - Math and Random

The Math library provides common functionality such as sine and cosine, maximum and minimum, and constants such as *pi* and *e*. Most of the functionality in the Math library is implemented as top-level functions.

To use the Math library in your app, import `dart:math`. The following examples use the prefix `math` to make clear which top-level functions and constants are from the Math library.

```
import 'dart:math' as math;
```

Trigonometry

The Math library provides basic trigonometric functions.

These methods use radians, not degrees!

```
// Cosine
assert(math.cos(math.PI) == -1.0);

// Sine
var degrees = 30;
var radians = degrees * (math.PI / 180);
// radians is now 0.52359.
var sinOf30degrees = math.sin(radians);

// Truncate the decimal places to 2.
assert(double.parse(sinOf30degrees.toStringAsPrecision(2)) == 0.5);
```

Maximum and Mininum

The Math library provides optimized `max()` and `min()` methods.

```
assert(math.max(1, 1000) == 1000);
assert(math.min(1, -1000) == -1000);
```

Math Constants

Find your favorite constants—*pi*, *e*, and more—in the Math library.

```
// See the Math library for additional constants.
print(math.E);      // 2.718281828459045
print(math.PI);     // 3.141592653589793
print(math.SQRT2); // 1.4142135623730951
```

Random Numbers

Generate random numbers with the Random class. You can optionally provide a seed to the Random constructor.

```
var random = new math.Random();
random.nextDouble(); // Between 0.0 and 1.0: [0, 1)
random.nextInt(10);  // Between 0 and 9.
```

You can even generate random booleans.

```
var random = new math.Random();
random.nextBool();  // true or false
```

More Information

Refer to the Math API docs for a full list of methods. Also see the API docs for num, int, and double.

dart:html - Browser-Based Apps

Use the dart:html library to program the browser, manipulate objects and elements in the DOM, and access HTML5 APIs. *DOM* stands for *Document Object Model*, which describes the hierarchy of an HTML page.

Other common uses of dart:html are manipulating styles (*CSS*), getting data using HTTP requests, and exchanging data using WebSockets (page 74). HTML5 (and dart:html) has many additional APIs that this section doesn't cover. Only web apps can use dart:html, not command-line apps.

To use the HTML library in your web app, import `dart:html`.

```
import 'dart:html';
```

Manipulating the DOM

To use the DOM, you need to know about *windows*, *documents*, *elements*, and *nodes*.

A Window object represents the actual window of the web browser. Each Window has a `document` property (a Document object), which points to the document currently loaded. The Window object also has accessors to various APIs such as IndexedDB (for storing data), `requestAnimationFrame()` (for animations), and more. In tabbed browsers, each tab has its own Window object.

With the Document object, you can create and manipulate Elements within the document. Note that the document itself is an element and can be manipulated.

The DOM models a tree of Nodes. These nodes are often elements, but they can also be attributes, text, comments, and other DOM types. Except for the root node, which has no parent, each node in the DOM has one parent and might have many children.

Finding elements

To manipulate an element, you first need an object that represents it. You can get this object using a query.

Find one or more elements using the top-level functions `query()` and `queryAll()`. You can query by ID, class, tag, name, or any combination of these. The CSS Selector Specification guide defines the formats of the selectors such as using a # prefix to specify IDs and a period (.) for classes.

The query() function returns the first element that matches the selector, while `queryAll()` returns a collection of elements that match the selector.

```
Element elem1 = query('#an-id');           // Find an element by id (an-id).
Element elem2 = query('.a-class');         // Find an element by class (a-class).
List<Element> elems1 = queryAll('div');    // Find all elements by tag (<div>).
List<Element> elems2 = queryAll('input[type="text"]'); // Find all text inputs.
```

```
// Find all elements with the CSS class 'class' inside of a <p>
// that is inside an element with the ID 'id'.
List<Element> elems3 = queryAll('#id p .class');
```

Manipulating elements

You can use properties to change the state of an element. Node and its subtype Element define the properties that all elements have. For example, all elements have `classes`, `hidden`, `id`, `innerHTML`, `style`, `text`, and `title` properties. Subclasses of Element define additional properties, such as the `href` property of AnchorElement.

Consider this example of specifying an anchor element in HTML:

```
<a id='example' href='http://example.com'>linktext</a>
```

This <a> tag specifies an element with an `href` attribute and a text node (accessible via a `text` property) that contains the string "linktext". To change the URL that the link goes to, you can use AnchorElement's `href` property:

```
query('#example').href = 'http://dartlang.org';
```

Often you need to set properties on multiple elements. For example, the following code sets the `hidden` property of all elements that have a class of "mac", "win", or "linux". Setting the `hidden` property to true has the same effect as adding `display:none` to the CSS.

```
<!-- Some HTML -->
<p>
  <a href="/downloads/linux" class="os linux">Download for Linux</a>
  <a href="/downloads/mac" class="os mac">Download for Mac</a>
  <a href="/downloads/win" class="os win">Download for Windows</a>
</p>

// Some Dart.

final osList = ['mac', 'win', 'linux'];

main() {
  var userOs = determineUserOs();

  for (var os in osList) {            // For each possible OS...
    bool shouldShow = (os == userOs); // Does this OS match the user's OS?
    for (var elem in queryAll('.$os')) { // Find all elements for this OS.
      elem.hidden = !shouldShow;      // Show or hide each element.
    }
  }
}
```

When the right property isn't available or convenient, you can use Element's `attributes` property. This property has the type AttributeMap, which implements a map (page 60) with keys that are strings (attribute names) and values that it automatically converts to strings. For a list of attribute names and their meanings, see the MDN Attributes page. Here's an example of setting an attribute's value.

```
elem.attributes['someAttribute'] = 'someValue';
```

Creating elements

You can add to existing HTML pages by creating new elements and attaching them to the DOM. Here's an example of creating a paragraph (<p>) element:

```
var elem = new ParagraphElement();
elem.text = 'Creating is easy!';
```

You can also create an element by parsing HTML text. Any child elements are also parsed and created.

```
var elem = new Element.html('<p>Creating <em>is</em> easy!</p>');
```

Note that elem is a ParagraphElement in the above example.

Attach the newly created element to the document by assigning a parent to the element. You can add an element to any existing element's children. In the following example, `body` is an element, and its child elements are accessible (as a List<Element>) from the `elements` property.

```
var elem = new ParagraphElement();
elem.text = "Don't forget to feed the llamas!";
document.body.elements.add(elem);
```

Adding, replacing, and removing nodes

Recall that elements are just a kind of node. You can find all the children of a node using the `nodes` property of Node, which returns a List<Node>. Once you have this list, you can use the usual List methods and operators to manipulate the children of the node.

To add a node as the last child of its parent, use the List `add()` method.

```
// Find the parent by ID, and add elem as its last child.
query('#inputs').nodes.add(elem);
```

To replace a node, use the Node `replaceWith()` method.

```
// Find a node by ID, and replace it in the DOM.
query('#status').replaceWith(elem);
```

To remove a node, use the Node `remove()` method.

```
// Find a node by ID, and remove it from the DOM.
query('#example').remove();
```

Manipulating CSS styles

CSS, or *cascading style sheets*, is used to define the presentation styles of DOM elements. You can change the appearance of an element by attaching ID and class attributes to it.

Each element has a `classes` field, which is a list. Add and remove CSS classes simply by adding and removing strings from this collection. For example, the following sample adds the `warning` class to an element.

```
var element = query('#message');
element.classes.add('warning');
```

It's often very efficient to find an element by ID. You can dynamically set an element ID with the `id` property.

```
var message = new DivElement();
message.id = 'message';
message.text = 'Please subscribe to the Dart mailing list.';
```

You can reduce the redundant text in this example by using method cascades:

```
var message = new DivElement()
    ..id = 'message'
    ..text = 'Please subscribe to the Dart mailing list.';
```

While using IDs and classes to associate an element with a set of styles is best practice, sometimes you want to attach a specific style directly to the element.

```
message.style
    ..fontWeight = 'bold'
    ..fontSize = '3em';
```

Handling events

To respond to external events such as clicks, changes of focus, and selections, add an event listener. You can add an event listener to any element on the page. Event dispatch and propagation is a complicated subject; research the details if you're new to web programming.

Add an event handler using `element.on.`*`event`*`.add(`*`function`*`)`, where *`event`* is the event name and *`function`* is the event handler.

For example, here's how you can handle clicks on a button.

```
// Find a button by ID and add an event handler.
query('#submitInfo').on.click.add((e) {
  // When the button is clicked, it runs this code.
  submitData();
});
```

Events can propagate up and down through the DOM tree. To discover which element originally fired the event, use `e.target`.

```
document.body.on.click.add((e) {
  var clickedElem = e.target;
  print('You clicked the ${clickedElem.id} element.');
});
```

To see all the events for which you can register an event listener, consult the API docs for ElementEvents and its subclasses. Some common events include:

- change
- blur
- keyDown
- keyUp
- mouseDown
- mouseUp

Using HTTP Resources with HttpRequest

Formerly known as XMLHttpRequest, the *HttpRequest* class gives you access to HTTP resources from within your browser-based app. Traditionally, AJAX-style apps make heavy use of HttpRequest. Use HttpRequest to dynamically load JSON data or any other resource from a web server. You can also dynamically send data to a web server.

The following examples assume all resources are served from the same web server that hosts the script itself. Due to security restrictions in the browser, the HttpRequest class can't easily use resources that are hosted on an origin that is different from the origin of the app. If you need to access resources that live on a different web server, you need to either use a technique called JSONP or enable CORS headers on the remote resources.

Getting data from the server

The `HttpRequest.get()` constructor is an easy way to get data from a web server.

```
import 'dart:html';
import 'dart:json';

onSuccess(HttpRequest request) {
  Map response = JSON.parse(request.responseText);
  String name = response['name'];
  String license = response['license'];
  print('The $name project uses the $license license.');
}

main() {
  // Request the data at 'data.json', a file in the same location as this page.
  var httpRequest = new HttpRequest.get('data.json', onSuccess);
}
```

The `onSuccess()` function runs when the data at the specified URI is successfully retrieved. In this case, we are dynamically loading a JSON file, whose contents are delivered in `request.responseText`. Information about the JSON API is in "dart:json - Encoding and Decoding Objects" (page 84).

The `HttpRequest.get()` constructor is great for simple cases, but you can also use the full API to handle more interesting cases. For example, you can capture errors and set arbitrary headers.

The general flow for using the full API of HttpRequest is as follows:

1. Create the HttpRequest object.
2. Open the URL with either `GET` or `POST`.
3. Attach event handlers.
4. Send the request.

A full example of handling errors follows:

```
import 'dart:html';
import 'dart:json';

loadEnd(HttpRequest request) {
  if (request.status != 200) {
    print('Uh oh, there was an error of ${request.status}');
    return;
  }

  Map response = JSON.parse(request.responseText);
  String name = response['name'];
  String license = response['license'];
  print('The $name project uses the $license license.');
}

main() {
  var dataUrl = 'data.json';
  var httpRequest = new HttpRequest();
  httpRequest.open('GET', dataUrl);
  httpRequest.on.loadEnd.add((e) => loadEnd(httpRequest));
  httpRequest.send();
}
```

Sending data to the server

HttpRequest can also send data to the server, using the HTTP method POST. For example, you might want to dynamically submit data to a form handler. Sending JSON data to a RESTful web service is another common example.

Submitting data to a form handler requires you to provide name-value pairs as URI-encoded strings. (Information about the URI API is in "dart:uri - Manipulating URIs" (page 86).) You must also set the `Content-type` header to `application/x-www-form-urlencode` if you wish to send data to a form handler.

```
import 'dart:html';
import 'dart:json';
import 'dart:uri';

String encodeMap(Map data) {
  return Strings.join(data.getKeys().map((k) {
    return '${encodeUriComponent(k)}=${encodeUriComponent(data[k])}';
  }), '&');
}

loadEnd(HttpRequest request) {
  if (request.status != 200) {
    print('Uh oh, there was an error of ${request.status}');
    return;
  } else {
    print('Data has been posted');
  }
}

main() {
  var dataUrl = '/registrations/create';
  var data = {'dart': 'fun', 'editor': 'productive'};
  var encodedData = encodeMap(data);

  var httpRequest = new HttpRequest();
  httpRequest.open('POST', dataUrl);
  httpRequest.setRequestHeader('Content-type',
                               'application/x-www-form-urlencoded');
  httpRequest.on.loadEnd.add((e) => loadEnd(httpRequest));
  httpRequest.send(encodedData);
}
```

Sending and Receiving Real-Time Data with WebSockets

A WebSocket allows your web app to exchange data with a server interactively—no polling necessary. A server creates the WebSocket and listens for requests on a URL that starts with **ws://**—for example, ws://127.0.0.1:1337/ws. The data transmitted over a WebSocket can be a string, a blob, or an ArrayBuffer. Often, the data is a JSON-formatted string.

To use a WebSocket in your web app, first create a WebSocket object, passing the WebSocket URL as an argument.

```
var webSocket = new WebSocket('ws://127.0.0.1:1337/ws');
```

Sending data

To send string data on the WebSocket, use the `send()` method.

```
sendMessage(String data) {
  if (webSocket.readyState == WebSocket.OPEN) {
    webSocket.send(data);
  } else {
    throw 'WebSocket not connected, message $data not sent';
  }
}
```

Receiving data

To receive data on the WebSocket, register a listener for message events.

```
webSocket.on.message.add((e) {
  receivedMessage((e as MessageEvent).data);
});
```

The message event handler receives a MessageEvent object. This object's `data` field has the data from the server. Here's an example of decoding a JSON string sent on a WebSocket, where the JSON string has two fields, "from" and "content".

```
// Called from the message listener like this: receivedMessage(e.data)
receivedMessage(String data) {
  Map message = JSON.parse(data);
  if (message['from'] != null) {
    print('Message from ${message['from']}: ${message['content']}');
  }
}
```

Handling WebSocket events

WebSocketEvents defines the WebSocket events your app can handle: open, close, error, and (as shown above) message. Here's an example of a method that creates a WebSocket object and handles message, open, close, and error events.

```
connectToWebSocket([int retrySeconds = 2]) {
  bool reconnectScheduled = false;
  webSocket = new WebSocket(url);

  scheduleReconnect() {
    print('web socket closed, retrying in $retrySeconds seconds');
    if (!reconnectScheduled) {
      window.setTimeout(() => connectToWebSocket(retrySeconds*2),
                                          1000*retrySeconds);
    }
    reconnectScheduled = true;
  }

  webSocket.on.open.add((e) {
    print('Connected');
  });
```

```
    webSocket.on.close.add((e) => scheduleReconnect());
    webSocket.on.error.add((e) => scheduleReconnect());

    webSocket.on.message.add((MessageEvent e) {
      _receivedEncodedMessage(e.data);
    });
  }
```

For an example of using WebSockets, see Chapter 5.

dart:isolate - Concurrency with Isolates

Dart has no shared-memory threads. Instead, all Dart code runs in isolates, which communicate via message passing. Messages are copied before they are received, ensuring that no two isolates can manipulate the same object instance. Because state is managed by individual isolates, no locks or mutexes are needed, greatly simplifying concurrent programming.

Isolate Concepts

To use isolates, you should understand the following concepts:

- No two isolates ever share the same thread at the same time. Within an isolate, callbacks execute one at a time, making the code more predictable.
- All values in memory, including globals, are available only to their isolate. No isolate can see or manipulate values owned by another isolate.
- The only way isolates can communicate with each other is by passing messages.
- Isolates send messages using SendPorts, and receive them using ReceivePorts.
- The content of a message can be any of the following:
  - A primitive value (null, num, bool, double, String)
  - An instance of SendPort
  - A list or map whose elements are any of the above, including other lists and maps
  - In special circumstances (page 78), an object of any type
- Each isolate has a ReceivePort, which is available as the `port` variable. Because all Dart code runs inside an isolate, even `main()` has access to a port object.
- When a web application is compiled to JavaScript, its isolates can be implemented as Web workers. When running in Dartium, isolates run in the VM.

- In the standalone VM, the `main()` function runs in the first isolate (also known as the *root isolate*). When the root isolate terminates, it terminates the whole VM, regardless of whether other isolates are still running. For more information, see "Keeping the root isolate alive" (page 79).

Using Isolates

To use an isolate, you import the `dart:isolate` library, spawn a new isolate, and then send and receive messages.

Spawning isolates

Any top-level function or static method[1] is a valid entry point for an isolate. The entry point should not expect arguments and should return void. It is illegal to use a function closure as an entry point to an isolate. Pass the entry point to `spawnFunction()`.

```
import 'dart:isolate';

runInIsolate() {
  print('hello from an isolate!');
}

main() {
  spawnFunction(runInIsolate);

  // Note: incomplete.
  // Use a ReceivePort (details below) to keep the root isolate alive
  // long enough for runInIsolate() to perform its work.
}
```

We plan to support spawning an isolate from code at a URI.

Sending messages

Send a message to an isolate via a SendPort. The `spawnFunction()` method returns a handle to the newly created isolate's SendPort.

To simply send a message, use `send()`.

```
import 'dart:isolate';

echo() {
  // Receive messages here. (See the next section.)
}

main() {
```

1. The dart2js compiler and the Dart VM do not yet support static methods as isolate entry points. For details, see *http://dartbug.com/3011*.

```
  var sendPort = spawnFunction(echo);
  sendPort.send('Hello from main');

  // Note: incomplete.
  // Use a ReceivePort (details below) to keep the root isolate alive
  // long enough for echo() to perform its work.
}
```

Sending any type of object

In special circumstances (such as when using `spawnFunction()` inside the Dart VM), it is possible to send any type of object to an isolate.[2] The object message is copied when sent.

Receiving messages

Use a ReceivePort to receive messages sent to an isolate. Obtain a handle to the default ReceivePort from the top-level `port` property. You can also create new instances of ReceivePort, if you want to route messages to different ports and callbacks.

Handle an incoming message with a callback function passed to the `receive()` method.

```
import 'dart:isolate';

echo() {
  port.receive((msg, reply) {
    print('I received: $msg');
  });
}

main() {
  var sendPort = spawnFunction(echo);
  sendPort.send('Hello from main');

  // Note: incomplete.
  // Use a ReceivePort (details below) to keep the root isolate alive
  // long enough for echo() to perform its work.
}
```

Receiving replies

Use the `call()` method on SendPort as a simple way to send a message and receive a reply. The `call()` method returns a Future for the reply.

```
import 'dart:isolate';

echo() {
  port.receive((msg, reply) {
    reply.send('I received: $msg');
```

2. Support for sending an arbitrary object to an isolate is not yet available when compiling to JavaScript.

```
  });
}

main() {
  var sendPort = spawnFunction(echo);
  sendPort.call('Hello from main').then((reply) {
    print(reply);    // I received: Hello from main
  });
}
```

Under the covers, the `call()` method creates and manages a SendPort and a ReceivePort, which are necessary for a call-and-response message exchange.

Keeping the root isolate alive

In the VM, an isolate continues to run as long as it has an open ReceivePort inside the isolate. If the `main()` function only starts other isolates, doing no work itself, you must keep the root isolate alive to keep the program alive.

To keep a root isolate alive, open a ReceivePort in the root isolate. When all the child isolates have finished their work, you can send a message to the root isolate to close its ReceivePort, thus stopping the program.

You can coordinate isolates with message passing, sending a message to inform the root isolate when a child isolate finishes. Here is an example:

```
import 'dart:isolate';

childIsolate() {
  port.receive((msg, replyTo) {
    print('doing some work');
    if (replyTo != null) replyTo.send('shutdown');
  });
}

main() {
  var sender = spawnFunction(childIsolate);
  var receiver = new ReceivePort();
  receiver.receive((msg, _) {
    if (msg == 'shutdown') {
      print('shutting down');
      receiver.close();
    }
  });
  sender.send('do work please', receiver.toSendPort());
}
```

In the above example, the child isolate runs to completion because the root isolate keeps a ReceivePort open. The root isolate creates a ReceivePort to wait for a shutdown message. The term shutdown is arbitrary; the ReceivePort simply needs to wait for some signal.

Once the root isolate receives a shutdown message, it closes the ReceivePort. With the ReceivePort closed and nothing else to do, the root isolate terminates, causing the app to exit.

More Information

See the API docs for the dart:isolate library, as well as for SendPort and ReceivePort.

dart:io - I/O for Command-Line Apps

The dart:io library provides APIs to deal with files, directories, processes, sockets, and HTTP connections. Only command-line apps can use dart:io—not web apps.

In general, the dart:io library implements and promotes an asynchronous API. Synchronous methods can easily block the event loop, making it difficult to scale server applications. Therefore, most operations return results via callbacks or Future objects, a pattern common with modern server platforms such as Node.js.

The few synchronous methods in the dart:io library are clearly marked with a Sync suffix on the method name. We don't cover them here.

Only command-line apps can import and use `dart:io`.

Files and Directories

The I/O library enables command-line apps to read and write files and browse directories. You have two choices for reading the contents of a file: all at once, or streaming. Reading a file all at once requires enough memory to store all the contents of the file. If the file is very large or you want to process it while reading it, you should use an InputStream, as described in "Streaming file contents" (page 81).

Reading a file as text

When reading a text file, you can read the entire file contents with `readAsText()`. When the individual lines are important, you can use `readAsLines()`. In both cases, a Future object is returned that provides the contents of the file as one or more strings.

```
import 'dart:io';

main() {
  var config = new File('config.txt');

  // Put the whole file in a single string.
  config.readAsText(Encoding.UTF_8).then((String contents) {
```

```
    print('The entire file is ${contents.length} characters long');
  });

  // Put each line of the file into its own string.
  config.readAsLines(Encoding.UTF_8).then((List<String> lines) {
    print('The entire file is ${lines.length} lines long');
  });
}
```

Reading a file as binary

The following code reads an entire file as bytes into a list of ints. The call to `readAsBytes()` returns a Future, which provides the result when it's available.

```
import 'dart:io';

main() {
  var config = new File('config.txt');

  config.readAsBytes().then((List<int> contents) {
    print('The entire file is ${contents.length} bytes long');
  });
}
```

Handling errors

Errors are thrown as exceptions if you do not register an explicit handler. If you want to capture an error, you can register a `handleException` handler with the Future object.

```
import 'dart:io';

main() {
  var config = new File('config.txt');
  Future readFile = config.readAsText();
  readFile.handleException((e) {
    print(e);
    // ...Other error handling goes here...
    return true; // We've handled the exception; no need to propagate it.
  });
  readFile.then((text) => print(text));
}
```

Streaming file contents

Use an InputStream to read a file, a little at a time. The `onData` callback runs when data is ready to be read. When the InputStream is finished reading the file, the `onClosed` callback executes.

```
import 'dart:io';

main() {
  var config = new File('config.txt');
  var inputStream = config.openInputStream();
```

```
  inputStream.onError = (e) => print(e);
  inputStream.onClosed = () => print('file is now closed');
  inputStream.onData = () {
    List<int> bytes = inputStream.read();
    print('Read ${bytes.length} bytes from stream');
  };
}
```

To decode an input stream from bytes into characters, wrap the InputStream with a StringInputStream. You can read the strings either as data becomes available or a line at a time.

Writing file contents

Use an OutputStream to write data to a file. Open a file for writing with `openOutputStream()` and declare a mode. Use `FileMode.WRITE` to completely overwrite existing data in the file, and `FileMode.APPEND` to add to the end.

```
import 'dart:io';

main() {
  var logFile = new File('log.txt');
  var out = logFile.openOutputStream(FileMode.WRITE);
  out.writeString('FILE ACCESSED ${new Date.now()}');
  out.close();
}
```

To write binary data, use `write(List<int> buffer)`.

Listing files in a directory

Finding all files and subdirectories for a directory is an asynchronous operation. The `list()` method returns a DirectoryLister, on which you can register callback handlers to be notified when a file is encountered (using `onFile`) or when a directory is encountered (using `onDir`).

```
import 'dart:io';

main() {
  var dir = new Directory('/tmp');

  DirectoryLister lister = dir.list(recursive:true); // Returns immediately.
  lister.onError = (e) => print(e);
  lister.onFile = (String name) => print('Found file $name');
  lister.onDir = (String name) => print('Found dir $name');
}
```

Other common functionality

The File and Directory classes contain other functionality, including but not limited to:

- Creating a file or directory: `create()` in File and Directory
- Deleting a file or directory: `delete()` in File and Directory
- Getting the length of a file: `length()` in File
- Getting random access to a file: `open()` in File

Refer to the API docs for File, Directory, and DirectoryLister for a full list of methods. Also see InputStream, StringInputStream, and OutputStream.

Besides the APIs discussed in this section, the dart:io library also provides APIs for processes, sockets, and web sockets.

For more examples of using dart:io, see Chapter 5.

HTTP Clients and Servers

The dart:io library provides classes that command-line apps can use for accessing HTTP resources, as well as running HTTP servers.

HTTP server

The HttpServer class provides the low-level functionality for building web servers. You can match request handlers, set headers, stream data, and more.

Because Dart is single threaded and has an event loop, the API design of HttpServer favors callbacks for handling events.

The following sample web server can return only simple text information. This server listens on port 8888 and address 127.0.0.1 (localhost), responding to requests for the path `/languages/dart`. All other requests are handled by the default request handler, which returns a response code of 404 (not found).

```
import 'dart:io';

main() {
  dartHandler(HttpRequest request, HttpResponse response) {
    print('New request');
    response.outputStream.writeString('Dart is optionally typed');
    response.outputStream.close();
  };

  var httpServer = new HttpServer();
  httpServer.addRequestHandler(
      (req) => req.path == '/languages/dart',
      dartHandler);

  httpServer.listen('127.0.0.1', 8888);
}
```

You can see a more comprehensive HTTP server in "The Server's Code" (page 116).

HTTP client

The HttpClient class helps you connect to HTTP resources from your Dart command-line or server-side application. You can set headers, use HTTP methods, and read and write data. The HttpClient class does not work in browser-based apps. When programming in the browser, use the HttpRequest class (page 72).

The HttpClient API, like the HttpServer API, is callback oriented. The general flow of events is as follows:

1. Create a new HttpClient.
2. Get the URL.
3. Register the `onResponse()` callback on the HttpClientConnection.
4. Register the `onData()` callback on the input stream.
5. Register the `onClosed()` callback on the HttpClientConnection.
6. Read data when available.
7. Shut down the HttpClient when you no longer want to create connections using it.

```
import 'dart:io';
import 'dart:uri';

main() {
  var httpClient = new HttpClient();
  var conn = httpClient.getUrl(new Uri('http://127.0.0.1:8888/languages/dart'));
  conn.onResponse = (HttpClientResponse resp) {
    var input = resp.inputStream;
    input.onData = () {
      var data = input.read();
      var text = new String.fromCharCodes(data);
      print(text);
    };
    input.onClosed = () => httpClient.shutdown();
  };
}
```

dart:json - Encoding and Decoding Objects

JSON is a simple text format for representing structured objects and collections. The JSON library decodes JSON-formatted strings into Dart objects, and encodes objects into JSON-formatted strings.

The Dart JSON library works in both web apps and command-line apps. To use the JSON library, import `dart:json`.

Decoding JSON

Decode a JSON-encoded string into a Dart object with `JSON.parse()`.

```
import 'dart:json';

main() {
  // NOTE: Be sure to use double quotes ("), not single quotes ('),
  // inside the JSON string. This string is JSON, not Dart.
  var jsonString = '''
  [
    {"score": 40},
    {"score": 80}
  ]
  ''';

  var scores = JSON.parse(jsonString);
  assert(scores is List);

  var firstScore = scores[0];
  assert(firstScore is Map);
  assert(firstScore['score'] == 40);
}
```

Encoding JSON

Encode a supported Dart object into a JSON-formatted string with `JSON.stringify()`.

Only objects of type int, double, String, bool, null, List, or Map can be encoded into JSON. List and Map objects are encoded recursively.

If any object that isn't an int, double, String, bool, null, List, or Map is passed to `stringify()`, the object's `toJson()` method is called. If `toJson()` returns an encodable value, that value is encoded in the object's place.

```
import 'dart:json';

main() {
  var scores = [
    {'score': 40},
    {'score': 80},
    {'score': 100, 'overtime': true, 'special_guest': null}
  ];

  var jsonText = JSON.stringify(scores);
  assert(jsonText == '[{"score":40},{"score":80},'
                     '{"score":100,"overtime":true,'
                     '"special_guest":null}]');
}
```

dart:uri - Manipulating URIs

The URI library provides functions to encode and decode strings for use in URIs (which you might know as *URLs*). These functions handle characters that are special for URIs, such as `&` and `=`.

Another part of the URI library is the Uri class, which parses and exposes the components of a URI—domain, port, scheme, and so on.

The URI library works in both web apps and command-line apps. To use it, import `dart:uri`.

Encoding and Decoding Fully Qualified URIs

To encode and decode characters *except* those with special meaning in a URI (such as `/`, `:`, `&`, `#`), use the top-level `encodeUri()` and `decodeUri()` functions. These functions are good for encoding or decoding a fully qualified URI, leaving intact special URI characters.

```
import 'dart:uri';

main() {
  var uri = 'http://example.org/api?foo=some message';
  var encoded = encodeUri(uri);
  assert(encoded == 'http://example.org/api?foo=some%20message');

  var decoded = decodeUri(encoded);
  assert(uri == decoded);
}
```

Notice how only the space between `some` and `message` was encoded.

Encoding and Decoding URI Components

To encode and decode all of a string's characters that have special meaning in a URI, including (but not limited to) `/`, `&`, and `:`, use the top-level `encodeUriComponent()` and `decodeUriComponent()` functions.

```
import 'dart:uri';

main() {
  var uri = 'http://example.org/api?foo=some message';
  var encoded = encodeUriComponent(uri);
  assert(encoded == 'http%3A%2F%2Fexample.org%2Fapi%3Ffoo%3Dsome%20message');

  var decoded = decodeUriComponent(encoded);
  assert(uri == decoded);
}
```

Notice how every special character is encoded. For example, `/` is encoded to `%2F`.

Parsing URIs

You can parse a URI into its parts with the `Uri()` constructor.

```
import 'dart:uri';

main() {
  var uri = new Uri('http://example.org:8080/foo/bar#frag');

  assert(uri.scheme == 'http');
  assert(uri.domain == 'example.org');
  assert(uri.path == '/foo/bar');
  assert(uri.fragment == 'frag');
  assert(uri.origin == 'http://example.org:8080');
}
```

See the Uri API docs for more URI components that you can get.

Building URIs

You can build up a URI from individual parts using the `Uri.fromComponents()` constructor.

```
import 'dart:uri';

main() {
  var uri = new Uri.fromComponents(scheme: 'http', domain: 'example.org',
                    path: '/foo/bar', fragment: 'frag');
  assert(uri.toString() == 'http://example.org/foo/bar#frag');
}
```

dart:utf - Strings and Unicode

The UTF library helps bridge the gap between strings and UTF-8/UTF-16/UTF-32 encodings.

The UTF library works in both web apps and command-line apps. To use the UTF library, import `dart:utf`.

Decoding UTF-8 Characters

Use `decodeUtf8()` to decode UTF8-encoded bytes to a Dart string.

```
import 'dart:utf';

main() {
  var string = decodeUtf8([0xc3, 0x8e, 0xc3, 0xb1, 0xc5, 0xa3, 0xc3, 0xa9,
                           0x72, 0xc3, 0xb1, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3,
                           0xae, 0xc3, 0xb6, 0xc3, 0xb1, 0xc3, 0xa5, 0xc4,
```

```
                          0xbc, 0xc3, 0xae, 0xc5, 0xbe, 0xc3, 0xa5, 0xc5,
                          0xa3, 0xc3, 0xae, 0xe1, 0xbb, 0x9d, 0xc3, 0xb1]);
  print(string); // 'Îñţérñåţîöñåļîžåţîð̃ñ'
}
```

Encoding Strings to UTF-8 Bytes

Use `encodeUtf8()` to encode a Dart string as a list of UTF8-encoded bytes.

```
import 'dart:utf';

main() {
  List<int> expected = [0xc3, 0x8e, 0xc3, 0xb1, 0xc5, 0xa3, 0xc3, 0xa9, 0x72,
                        0xc3, 0xb1, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3, 0xae, 0xc3,
                        0xb6, 0xc3, 0xb1, 0xc3, 0xa5, 0xc4, 0xbc, 0xc3, 0xae,
                        0xc5, 0xbe, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3, 0xae, 0xe1,
                        0xbb, 0x9d, 0xc3, 0xb1];

  List<int> encoded = encodeUtf8('Îñţérñåţîöñåļîžåţîð̃ñ');

  assert(() {
    if (encoded.length != expected.length) return false;
    for (int i = 0; i < encoded.length; i++) {
      if (encoded[i] != expected[i]) return false;
    }
    return true;
  });
}
```

Other Functionality

The UTF library can decode and encode UTF-16 and UTF-32 bytes. The library can also convert directly to and from Unicode code points and UTF8-encoded bytes. For details, see the API docs for the UTF library.

dart:crypto - Hash Codes and More

The Dart crypto library contains functions useful for cryptographic applications, such as creating cryptographic hashes and generating hash-based message authentication codes.

The crypto library works in both web apps and command-line apps. To use the crypto library, import `dart:crypto`.

Generating Cryptographic Hashes

With the crypto library, you can use SHA256, SHA1, or MD5 objects to generate hashes (also known as *digests* or *message digests*). We recommend using SHA256, but we have included SHA1 and MD5 for compatibility with older systems. All these types inherit from Hash, which defines the Dart interface for cryptographic hash functions.

```
import 'dart:crypto';

main() {
  var sha256 = new SHA256();
  var digest = sha256.update('message'.charCodes()).digest();
  var hexString = CryptoUtils.bytesToHex(digest);
  assert(hexString ==
      'ab530a13e45914982b79f9b7e3fba994cfd1f3fb22f71cea1afbf02b460c6d1d');
}
```

If the message content changes, the digest value also changes (with a very high probability).

Generating Message Authentication Codes

Use a hash-based message authentication code (HMAC) to combine a cryptographic hash function with a secret key.

```
import 'dart:crypto';
main() {
  var hmac = new HMAC(new SHA256(), 'secretkey'.charCodes());
  var hmacDigest = hmac.update('message'.charCodes()).digest();
  var hmacHex = CryptoUtils.bytesToHex(hmacDigest);
  assert(hmacHex ==
      '5c3e2f56de9411068f675ef32ffa12735210b9cbfee2ba521367a3955334a343');
}
```

If either the message contents or key changes, the digest value also changes (with a very high probability).

Generating Base64 Strings

You can represent binary data as a character string by using the Base64 encoding scheme. Use the `CryptoUtils.bytesToBase64()` utility method to convert a list of bytes into a Base64-encoded string.

```
import 'dart:crypto';
import 'dart:io';

main() {
  var file = new File('icon.ico');
  var bytes = file.readAsBytesSync();
  var base64 = CryptoUtils.bytesToBase64(bytes);
```

```
  assert(base64 ==
    'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38G'
    'IAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==');
}
```

Summary

This chapter introduced you to the most commonly used functionality in Dart's built-in libraries. You can use the pub tool, discussed in the next chapter, to install additional Dart libraries.

CHAPTER 4

Tools

Dart provides several tools to help you write and deploy your web and command-line apps.

pub: The Dart package manager
: Download and install packages of libraries.

Dart Editor (page 93)
: Edit, run, and debug web and command-line apps.

Dartium: Chromium with the Dart VM (page 103)
: Run Dart web apps. This is a special build of Chromium (the project behind Google Chrome).

dart2js: The Dart-to-JavaScript compiler (page 105)
: Convert your web app to JavaScript, so it can run in non-Dartium browsers.

dart: The standalone Dart VM (page 106)
: Run your command-line apps—server-side scripts, programs, servers, and any other apps that don't run in a browser.

All of these tools are in the Dart Editor bundle, since the editor uses Dartium and the other tools. You can also download Dartium separately, and you can download an SDK that includes dart2js, dart, and pub. See the Downloads page for links and details.

pub: The Dart Package Manager

You can use use the *pub* tool (`$DART_SDK/bin/pub`) to manage Dart packages. A Dart package is simply a directory containing any number of Dart libraries and a list of the library dependencies. A package can also contain resources for its libraries, such as documentation, tests, and images. If your app uses one or more packages, then your app itself must be a package.

A package can live anywhere. Some packages are distributed in the Dart SDK, under the `pkg` directory. Others are on GitHub. We plan to publish packages at pub.dartlang.org, and we hope you will, too.

To use a library that's in a Dart package, you need to do the following:

1. Create a pubspec (a file that lists package dependencies).
2. Use pub to install the package.
3. Import the library.

Creating a Pubspec

To use a package, your application must define a pubspec that lists dependencies and their download locations. The pubspec is a file named `pubspec.yaml`, and it must be in the top directory of your application.

Here is an example of a pubspec that specifies the locations of two packages. First, it points to the js package that's hosted on pub.dartlang.org, and then it points to the intl package in the Dart SDK.

```
name: my_app
dependencies:
  js:
    hosted: js
  intl:
    sdk: intl
```

For details, see the pubspec documentation and the documentation for the packages you're interested in using.

Installing Packages

Once you have a pubspec, you can run `pub install` from the top directory of your application.

```
cd my/app
$DART_SDK/bin/pub install
```

This command determines which packages your app depends on, and puts them in a central cache. For git dependencies, pub clones the git repository. For hosted dependencies, pub downloads the package from pub.dartlang.org. Transitive dependencies are included, too. For example, if the js package is dependent on the unittest package, the pub tool grabs both the js package and the unittest package.

Finally, pub creates a `packages` directory (under your app's top directory) that has links to the packages that your app depends on.

Importing Libraries from Packages

To import libraries found in packages, use the `package:` prefix.

```
import 'package:js/js.dart' as js;
import 'package:intl/intl.dart';
```

The Dart runtime takes everything after `package:` and looks it up within the `packages` directory for your app.

More Information

Run *`$DART_SDK`*`/bin/pub --help` for a list of commands. For more information about pub, see the pub documentation.

Dart Editor

We already introduced Dart Editor in "Up and Running" (page 5). Here are some more tips on using Dart Editor, with information such as specifying a browser (page 100) and compiling to JavaScript (page 102). If you run into a problem, see Troubleshooting Dart Editor. Dart Editor is updated frequently, so it probably looks different from what you see here. For the latest information, see the Dart Editor documentation.

Viewing Samples

The Welcome page of Dart Editor (Figure 1-2) displays a few samples. To open a sample and look at its source code, click the sample's image.

If you don't see the Welcome page, you probably closed it. Get it back with **Tools** > **Welcome Page**.

Managing the Files View

The Files view shows the files that implement the libraries included in Dart, as well as all the apps that you create or open.

Adding apps

Here's how to open an app, which makes it appear in your Files view:

1. Go to the **File** menu, and choose **Open Folder...**. Or use the keyboard shortcut (**Ctrl+O** or, on Mac, **Cmd+O**).
2. Select the directory that contains the app's files, and click **Open**.

The directory and all its files appear in the Files view.

Removing apps

You can remove an app from the Files view, either with or without deleting its files.

Right-click (or Ctrl+click) the directory and choose **Delete**. If you want to delete the app's files permanently, then in the dialog that comes up, choose **Delete projects on disk**.

Creating Apps

It's easy to create a simple web or command-line app from scratch:

1. Click the New Application button (at the upper left of Dart Editor). Alternatively, choose **File > New Application...** from the Dart Editor menu. A dialog appears.
2. Type in a name for your application—for example, `HelloWeb`. If you don't like the default directory, type in a new location or browse to choose the location.
3. If you're creating a web app, select **Generate content for a basic web app**. If you want to use the pub package manager, select **Add pub support**. Then click **Finish** to create a directory with initial files for the app.

   A default Dart file appears in the Edit view, and its directory appears in the Files view. Your Dart Editor window should look something like Figure 4-1.

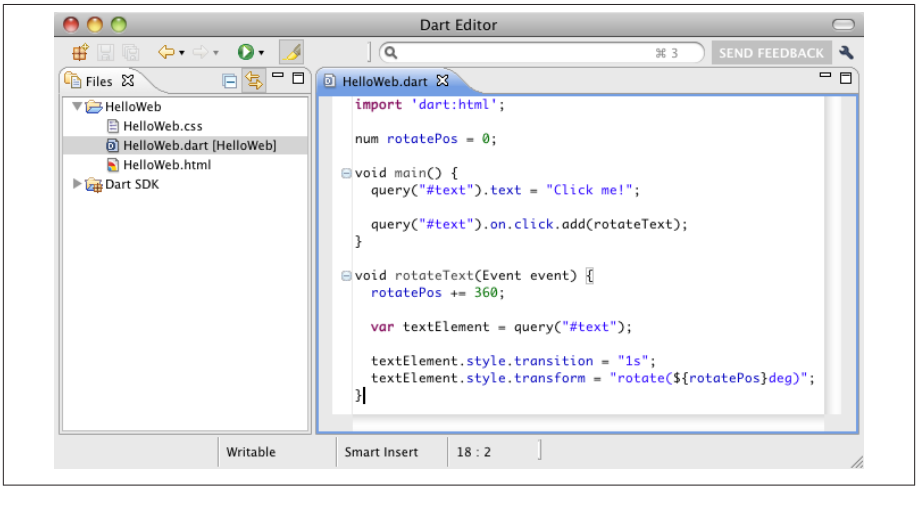


Figure 4-1. A new app, pre-filled with basic, runnable code

Editing Apps

Dart Editor provides the basic editing functionality you'd expect, plus features such as Dart code completion, API browsing, support for refactoring, and the ability to search multiple files.

Using autocomplete

Autocomplete suggestions look something like Figure 4-2. They appear when you either:

- Type a class or variable name, and then type a period.

  For example, type `document.` or `Date.` and pause a moment. Once the suggestions appear, continue typing to pare down the list.
- Type **Ctrl+Space**.

  For example, type `Dat`, then Ctrl+Space to see a list of classes that start with "Dat".

When the suggestions come up, you can click, type, or use the arrow keys to select the one you want. Press Enter or Return to choose a suggestion, or Esc to dismiss the panel.

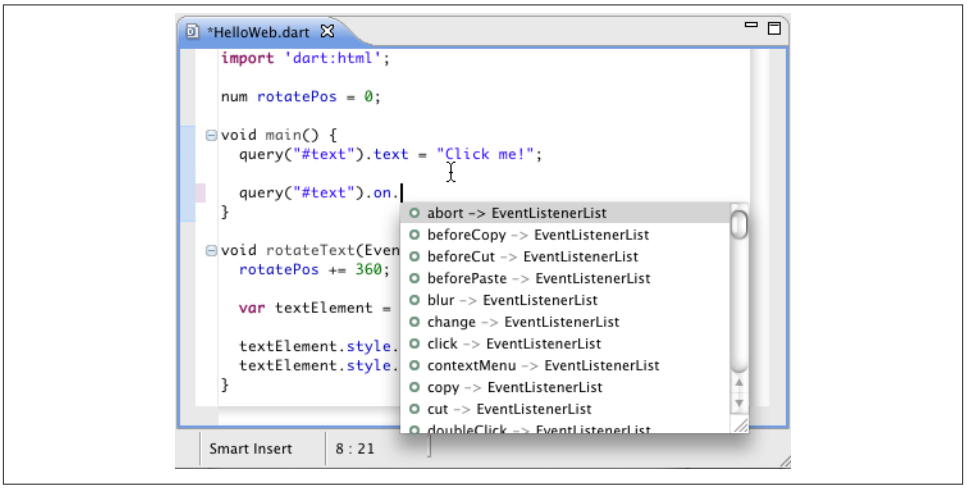


Figure 4-2. Autocomplete suggestions

Browsing APIs

With Dart Editor you can easily find where APIs are declared. You can also outline the contents of a Dart file.

Finding where an API is declared. Use this feature to go to the declaration of an API item —variable, method, type, library, and so on—either within the same .dart file or in another file.

1. In the Edit view of a Dart file, hold down the **Control** key (**Command** key on Mac) and move the mouse over the source code. As Figure 4-3 shows, any API item under your cursor is underlined.

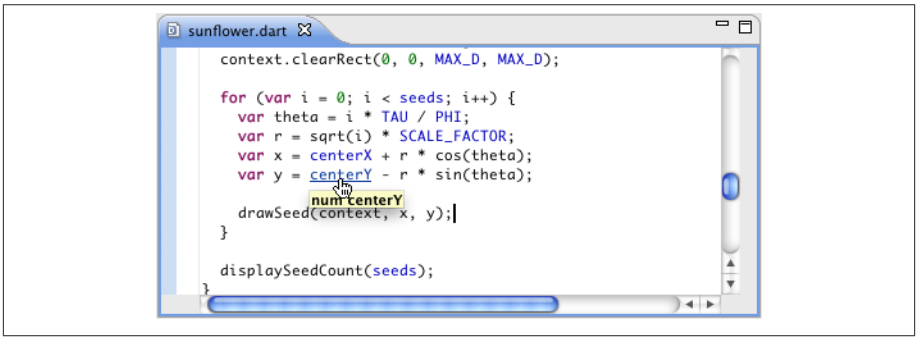


Figure 4-3. Control or Command while hovering over API items

2. Continuing to press Control or Command, hover over an underlined string, and click.

   The editor displays the file that declares the item. For example, if you Command-click cos, the file that declares the `cos()` function appears.

Outlining a file's contents. Either press **Alt+O** (**Option+O** on Mac) or right-click and choose **Quick Outline**.

A panel comes up displaying the classes, methods, and fields declared in the current Dart file. For example, the outline for the Sunflower sample's `sunflower.dart` file looks something like Figure 4-4.

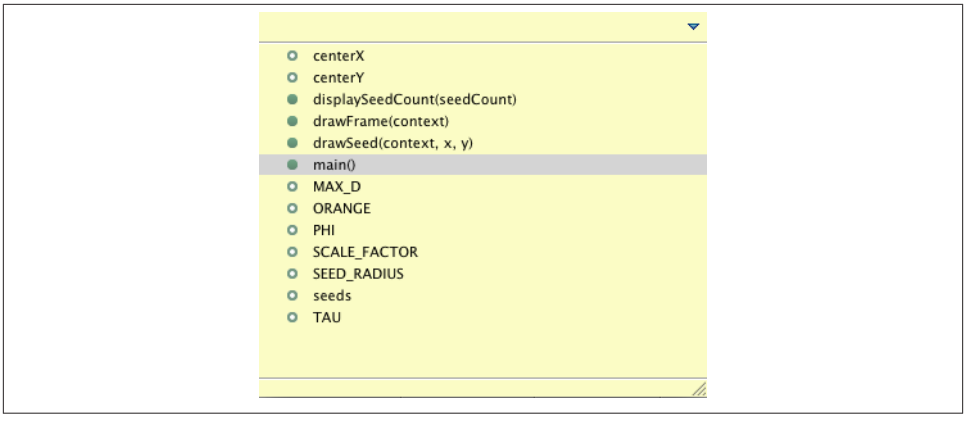


Figure 4-4. The Outline panel for the Sunflower sample

You can reduce the size of the list by typing one or more characters. For example, if you type **c**, only the **center** variables appear.

If you choose an item from the list—for example, **centerX**— the editor scrolls to its declaration.

Alternatively, add a more permanent outline view by choosing **Tools** > **Outline**.

Finding where an API is used. Use the Callers view to find where an API item—such as a function—is used. To bring up the view, click the item's name, right-click, and choose **Find Callers** (or type Ctrl+Alt+H).

To see how a caller itself is called, click its arrow in the Callers view. For example, Figure 4-5 shows that two functions call the `updateTime()` method.

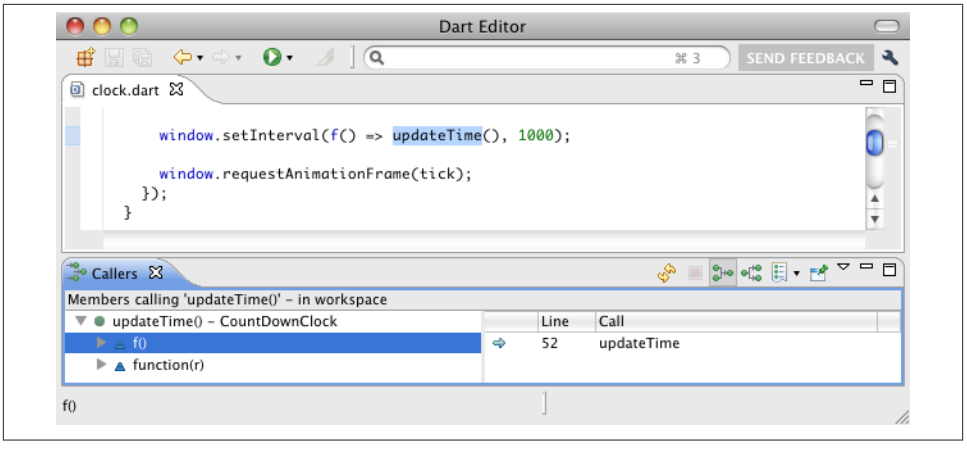


Figure 4-5. The Callers view

Once the Callers view is visible, you can click the Show Callee Hierarchy button to make the view show *callees*—the functions called by the function you're inspecting.

Refactoring

To change the name of an item throughout your code, put the cursor within (or double-click) the item's name in the Edit view, and choose **Edit > Rename...** or right-click and choose **Rename...**.

You can rename almost anything—local variables, function parameters, fields, methods, types, top-level functions, library prefixes, top-level compilation units, and more. An example of renaming a top-level compilation unit is changing the name of a file that's sourced by a library.

Searching

The search field at the upper right of the Dart Editor window is an easy way to go directly to:

- Types
- Files
- Text inside of files

The scope of a text search is every file in your Files view. Results for text searches come up in a Search view. Within that view, double-click a file to see it in the Edit view. All occurrences of the search string in the Edit view are highlighted.

Running Apps

To run any Dart app, click Dart Editor's Run button  while any file in that app is selected. If you're working on a web app, Dart Editor brings up a browser window that displays the app's HTML page, with the app's code running inside it.

When you run a web application using Dart Editor, by default the app uses the copy of Dartium that's included in the Dart Editor download, with the Dart code executing directly in the browser. If your launch configuration specifies a browser, then Dart Editor uses dart2js (page 105) to compile the Dart code to JavaScript that executes in the browser.

Specifying launch configurations

Use **Run** > **Manage Launches** to specify as many launch configurations as you like.

For web apps, you can specify the following:

- HTML file or URL to open
- browser (JavaScript only)
- arguments to pass to the browser; for example, `--allow-file-access-from-files`
- debugging enabled (Dartium only)
- checked mode (Dartium only)
- whether to show the browser's stdout and stderr output (Dartium only; useful for diagnosing Dartium crashes)

For example, a web app might have a launch configuration for Dartium and several more configurations corresponding to additional browsers you want to test.

You can specify the following for command-line apps:

- .dart file to execute
- arguments to pass to the app
- checked mode
- heap size

Running in production mode

By default, apps run in checked mode. To run in production mode instead, disable checked mode in your app's launch configuration:

1. Run your app at least once, so that it has a launch configuration.
2. Choose **Run** > **Manage Launches**, or click the little arrow to the right of the Run button .

3. In the Manage Launches dialog, find a launch configuration for your app. Click it if it isn't already selected.
4. Unselect **Run in checked mode** (see Figure 4-6).

Figure 4-6. To run in production mode, unselect checked mode

5. Click **Apply** to save your change, or **Run** to save it and run your app.

For details about checked mode and production mode, see "Runtime Modes" (page 13).

Specifying a browser

To specify the browser in which the your app runs:

1. Choose **Run** > **Manage Launches**, or click the little arrow to the right of the Run button. The Manage Launches dialog appears (see Figure 4-7). On the left side of the dialog is a list of all launch configurations that you've created or that were automatically created for you. On the right is information about the currently selected launch configuration.

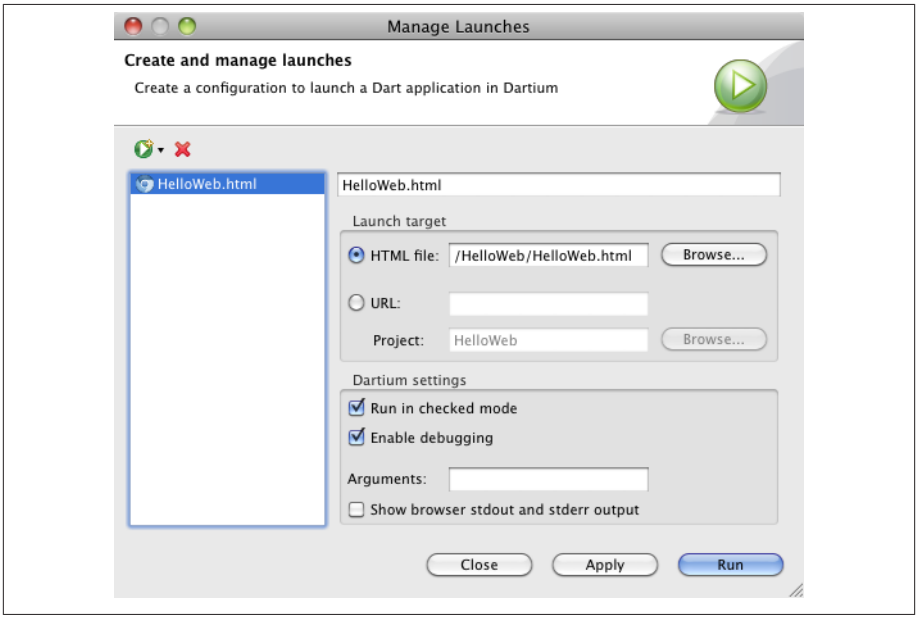


Figure 4-7. The Manage Launches dialog

2. In the Manage Launches dialog, click the Create new launch button, and choose **New web launch: JavaScript**.

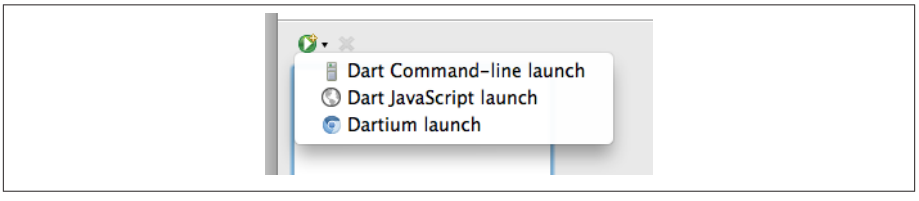


Figure 4-8. Creating a new web launch

3. Enter configuration information:
   - Give the configuration a name that identifies the app, the browser, and anything else important. *Example:* HelloWeb.html in Firefox
   - Specify the HTML file or URL to open. *Example:* /HelloWeb/HelloWeb.html
   - Unless you want to use the default system browser, unselect **Use default system browser** and specify the location of the browser you want to use. *Example:* /Applications/Firefox.app
4. Click **Apply** to save your changes, or **Run** to save your changes and launch the app.

Now that you've set up the launch, you can choose it any time from the Run button. Your app will be automatically compiled to JavaScript each time you run it.

Debugging Apps

You can debug both command-line and web apps with Dart Editor. Debugging must be enabled in your launch configuration (which it is, by default).

Some tips for debugging:

- Set breakpoints by double-clicking in the left margin of the Edit view.
- Use the Debugger view to view your app's state and control its execution. By default, the Debugger view is to the right of the Edit view.
- To see the values of variables, mouse over the variable or look in the Debugging view's Call Stack.
- Because everything in Dart is an object and operators are really methods, you'll probably use Step Return (F7) more than you'd expect to climb out of Dart libraries.

- To debug web apps, you use Dart Editor with Dartium (Figure 4-9). While you're debugging, Dart Editor takes the place of the Dartium console. For example, Dart Editor displays the output of `print()` statements.

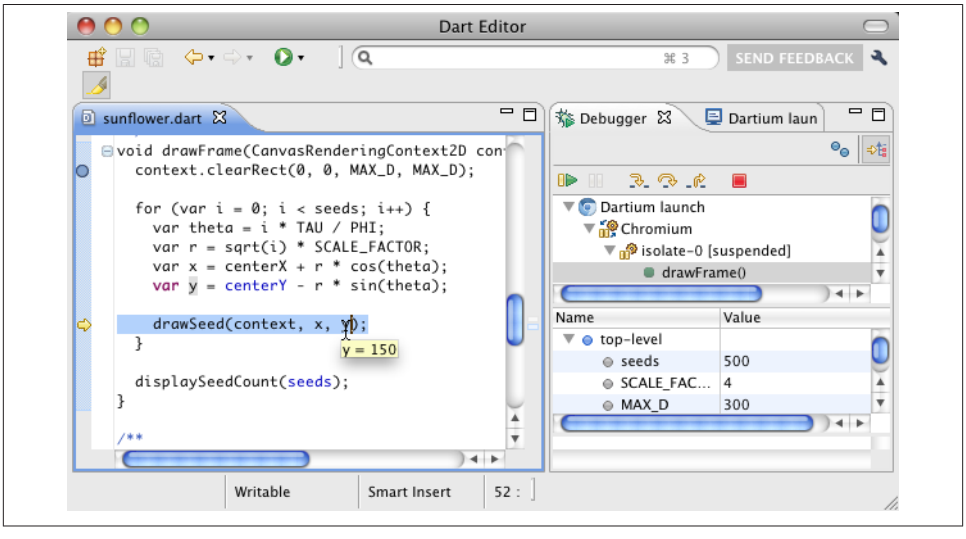


Figure 4-9. Using Dart Editor to debug the Sunflower sample app

Compiling to JavaScript

You might not need to do anything to compile your code to JavaScript. When you run an app using a launch configuration that specifies a browser (page 100), Dart Editor automatically compiles the app to JavaScript before executing it in the browser.

However, you can also compile Dart code to JavaScript without running the app. Just choose **Tools > Generate JavaScript**. Another option is using dart2js from the command line (see "dart2js: The Dart-to-JavaScript Compiler" (page 105)).

Other Features

Dart Editor has many additional features, including doc generation, customization, and keyboard alternatives.

Generating documentation with dartdoc

Use the **Tools > Generate Dartdoc** command to generate HTML documentation from Dart code. For information on supplying text for the documentation, see "Documentation Comments" (page 51).

Customizing Dart Editor

You can customize the editor's font, margins, key bindings, and more using the Preferences dialog. To bring up the dialog, choose **Tools** > **Preferences** (on Mac: **Dart Editor** > **Preferences**).

You can also customize which views you see in Dart Editor, as well as their size and position. To add views, use the **Tools** menu. To remove a view, click its **X**. To move a view, drag its tab to a different position, either within or outside of the Dart Editor window. To resize a view, drag its edges.

Keyboard alternatives

To get a list of all keyboard alternatives, choose **Help** > **Key Assist** (Figure 4-10).

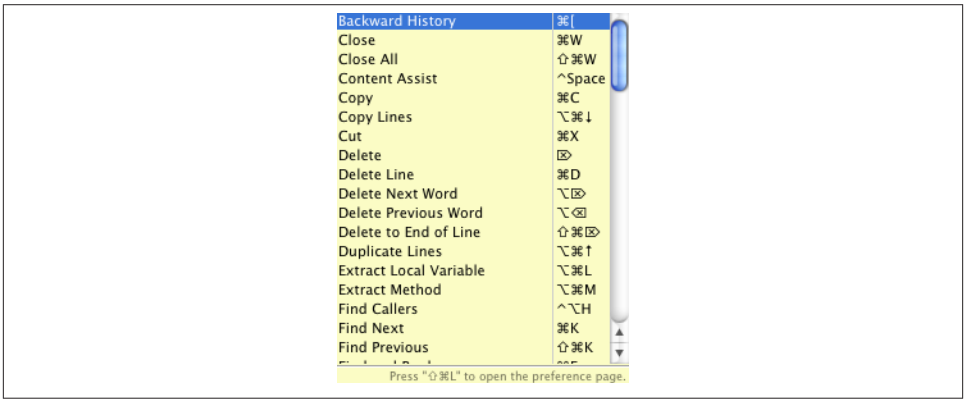
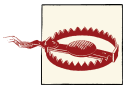


Figure 4-10. Help > Key Assist

Dartium: Chromium with the Dart VM

This section tells you how to get and use Dartium, a Chromium-based browser that includes the Dart VM. This browser can execute Dart web apps directly, so you don't have to compile your code to JavaScript until you're ready to test on other browsers.



This browser is a technical preview, and it might have security and stability issues. **Do not use Dartium as your primary browser!**

Downloading and Installing the Browser

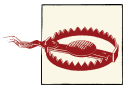
If you have an up-to-date version of Dart Editor, you already have Dartium.

If you don't have Dart Editor or want a different version of Dartium, you can get it separately from the Downloads page. The Dartium binary does expires after a while. When that happens, you'll need to download a new copy if you want to continue using Dartium.

You don't usually need to do anything special to install Dartium: just unarchive the ZIP file. If you want Dart Editor to launch a particular copy of Dartium, then put that copy inside the `dart-sdk` directory of your Dart installation directory (see "Step 1: Download and Install the Software" (page 5)), replacing the original copy of Chromium.

Launching the Browser

To launch Dartium, navigate to its directory in your finder, and double-click the Chromium executable file. Or use Dart Editor as described in "Running Apps" (page 99) or the command line as described in "Launching from the Command Line" (page 105).



If you already use Chromium: If another version of Chromium is open, then you could have a profile conflict. To avoid this, you can open Dartium or Chromium from the command line with the --user-data-dir flag.

Filing Bugs

If you find a bug in Dartium, create an issue in the Dart project and use the Dartium bug template.

Linking to Dart Source

Use a script tag with a type `application/dart` to link to your Dart source file. Example:

```
<!DOCTYPE html>
<html>
  <body>
    <script type="application/dart" src="app.dart"></script>

    <!-- bootstraps the Dart VM and handles non-Dart browsers -->
    <script type="text/javascript"
      src="http://dart.googlecode.com/svn/trunk/dart/client/dart.js">
    </script>
  </body>
</html>
```

Dart Editor automatically adds both the `application/dart` script tag and the bootstrap JavaScript tag into the project's main HTML file.

Detecting Dart Support

The above example uses a bootstrap script that takes care of turning on the Dart VM, as well as compatibility with non-Dart browsers. Instead of using the `dart.js` script, you can manually include the necessary JavaScript code.

To start the Dart VM, use the JavaScript function `navigator.webkitStartDart()`. For example:

```
// In JavaScript code:
if (!navigator.webkitStartDart) {
  // No native Dart support.
  window.addEventListener("DOMContentLoaded", function (e) {
      // ...Fall back to compiled JS...
    }
  }, false);
} else {
  navigator.webkitStartDart();
}
```

Launching from the Command Line

Because Dartium is based on Chromium, all Chromium flags should work. In some cases, you might want to specify Dart-specific flags so that you can tweak the embedded Dart VM's behavior. For example, while developing your web app, you might want the VM to perform as many checks as possible. To achieve that, you can enable checked mode (the VM's --enable-type-checks flag) and assertion checks (--enable-asserts flag).

On Linux, you can specify flags by starting Dartium as follows:

```
DART_FLAGS='--enable-type-checks --enable-asserts' path/chrome
```

On Mac:

```
DART_FLAGS='--enable-type-checks --enable-asserts'              \
  path/Chromium.app/Contents/MacOS/Chromium
```

Or (also on Mac):

```
DART_FLAGS='--enable-type-checks --enable-asserts'              \
   open path/Chromium.app
```

You can see the command-line flags and executable path of your current Chromium-based browser by going to `chrome://version`.

dart2js: The Dart-to-JavaScript Compiler

You can use the *dart2js* tool to compile Dart code to JavaScript. Dart Editor (page 102) uses dart2js behind the scenes whenever Dart Editor compiles to JavaScript.

Basic Usage

Here's an example of compiling a Dart file to JavaScript:

```
$DART_SDK/bin/dart2js test.dart
```

This command produces a `.js` file that contains the JavaScript equivalent of your Dart code.

Options

Common command-line options for dart2js include:

`-o<file>`
: Generate the output into *<file>*.

`-c`
: Insert runtime type checks, and enable assertions (checked mode).

`-h`
: Display help (add `-v` for information about all options).

dart: The Standalone VM

You can use the *dart* tool (`bin/dart`) to run Dart command-line apps such as server-side scripts, programs, and servers. During development, you also have the option to run command-line apps using Dart Editor (page 99).

Basic Usage

Here's an example of running a Dart file on the command line:

```
$DART_SDK/bin/dart test.dart
```

Enabling Checked Mode

Dart programs run in one of two modes: checked or production. By default, the Dart VM runs in production mode. We recommend that you enable checked mode for development and testing.

In checked mode, assignments are dynamically checked, and certain violations of the type system raise exceptions at run time. In production mode, static type annotations have no effect.

Assert statements are also enabled in checked mode. An assert statement (page 32) checks a boolean condition, raising an exception if the condition is false. Assertions do not run in production mode.

You can run the VM in checked mode with a command-line flag:

```
$DART_SDK/bin/dart --checked test.dart
```

Additional Options

Print all the command-line options with `--print-flags`:

```
$DART_SDK/bin/dart --print-flags
```

Summary

This chapter covered the most commonly used Dart tools. All of them are available in the Dart Editor download, but you can also download Dartium or the SDK separately.

CHAPTER 5

Walkthrough: Dart Chat

This chapter points out some of the useful and fun features of Dart that we used to build Dart Chat, a client-server app. If you'd like step-by-step instructions on building Dart Chat, you might be interested in our code lab.

Figure 5-1 shows the chat client executing in a Dartium window. Each copy of the chat client can send messages to the chat server, which forwards those messages to the other chat clients.

How to Run Dart Chat

The easiest way to run the Dart Chat client and server apps is to open them in Dart Editor.

1. Download the Dart Chat source code from GitHub.
2. In Dart Editor, use **File > Open Folder...**, to open the **finished** directory of the Dart Chat source code.
3. Select `chat-server.dart`, and then click the Run button  . A view named chat-server appears in Dart Editor, displaying debugging output for the server.
4. Select `client/chat-client.dart`, and then click the Run button  . Dartium launches, if necessary, and displays a Dart Chat tab.
5. To create another copy of the chat client, go to the Dart Chat tab in Dartium. Right-click the tab, and choose **Duplicate**.

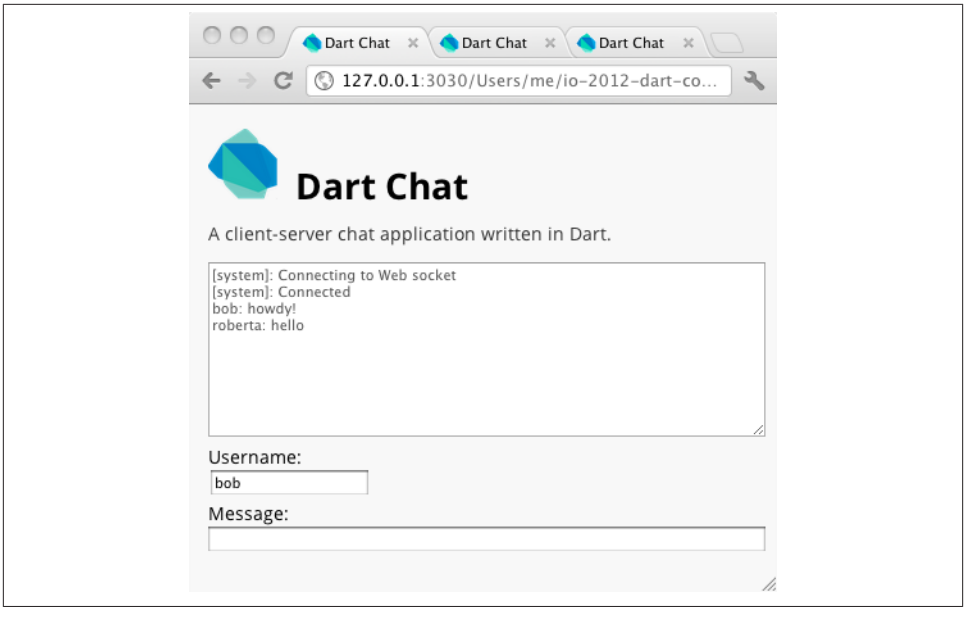


Figure 5-1. Multiple chat clients can use the chat server to talk

How Dart Chat Works

The chat server and client are simple. The chat server is an HTTP server that provides a WebSocket. The chat client uses that WebSocket for a bi-directional communication channel with the server. The client sends chat messages to the sever over the WebSocket, and the server relays those messages to all other connected clients.

As Figure 5-2 shows, the server starts things off by listening for requests to `ws://127.0.0.1:1337/ws`. Chat clients then connect to that URL.

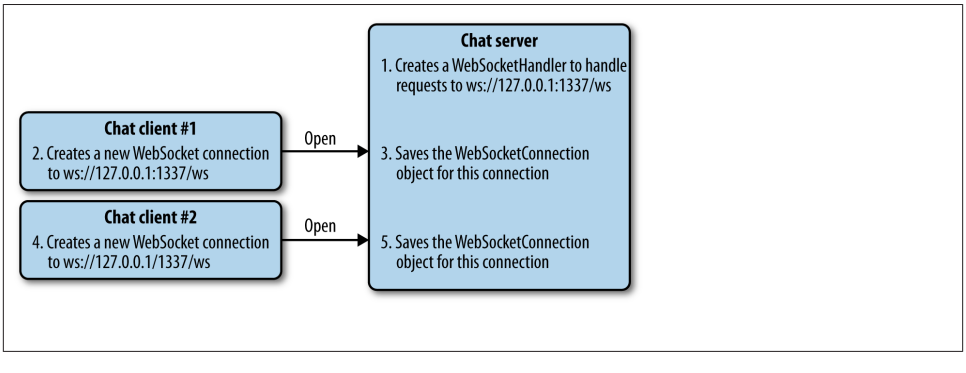


Figure 5-2. Chat clients connect to a web socket created by the chat server

The real communication between client and server happens when the user enters a message. As Figure 5-3 shows, the chat client sends a JSON-encoded version of the message to the server. The server then forwards the message to every client except the one that sent it.

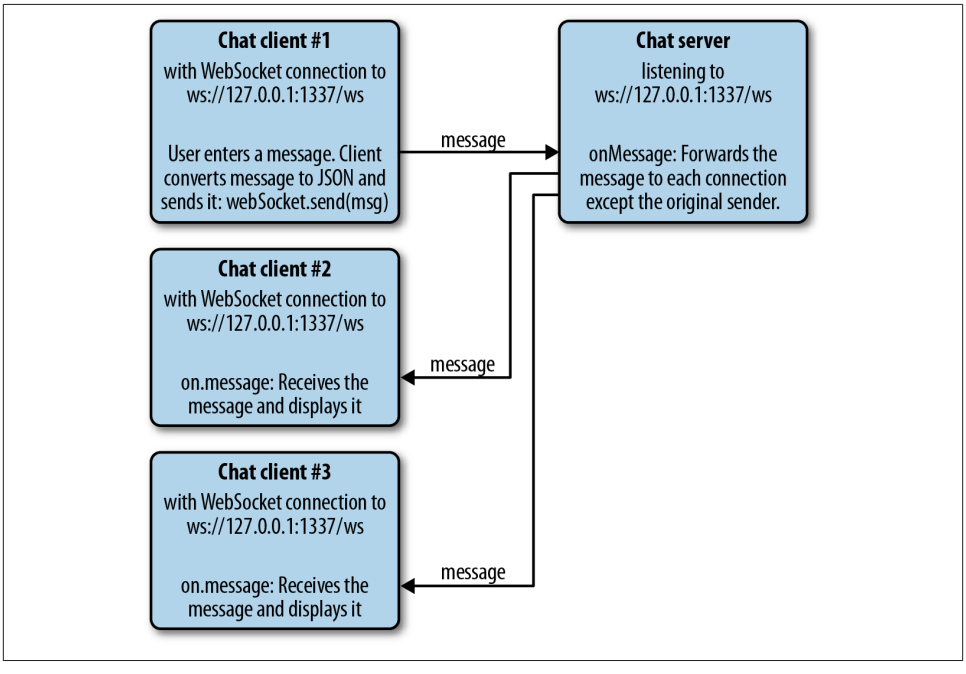


Figure 5-3. A chat client uses the server to send a message to other chat clients

The chat server implements an HTTP server to listen for WebSocket requests. The HTTP server can also serve static files from the client directory—for example, http://127.0.0.1:1337/chat-log.txt shows the file that's at client/chat-log.txt.

The client code is split between HTML (page structure), CSS (page look), and JavaScript (logic and behavior). That's typical of web clients.

The twist is that this client's JavaScript code is produced from Dart code, thanks to the dart2js compiler. Any modern browser can run this JavaScript code. Dartium (and any other browsers that support Dart) can run either the JavaScript code or the original Dart code.

The Client's HTML Code

The main elements in the client UI are two text fields (with the IDs "chat-username" and "chat-message") and a status area (ID: "chat-display").

```
<!-- In client/index.html: -->
<textarea id="chat-display" rows="10" disabled></textarea>
...
<input id="chat-username" name="chat-username" type="text">
...
<input id="chat-message" name="chat-message" type="text" disabled
    value="enter message...">
```

Near the bottom of client/index.html, a couple of <script> tags tell the browser to execute the client's Dart or JavaScript code:

```
<script type="application/dart" src="chat-client.dart"></script>
<script src="dart.js"></script>
```

The first line works in browsers that have an embedded Dart VM and so can execute Dart code; currently, only Dartium qualifies. The second line is important for every other browser. It executes dart.js, which is a standard script that converts all Dart <script> tags to use foo.dart.js instead of foo.dart, with the assumption that foo.dart.js is a JavaScript version of foo.dart. For non-Dartium browsers, dart.js changes the first <script> tag to the following:

```
<!-- Inserted by dart.js for non-Dartium browsers -->
<script src="chat-client.dart.js"></script>
```

The script contents run when the browser has loaded the HTML and constructed its DOM (document object model).

You can get dart.js from the Dart project. See "dart2js: The Dart-to-JavaScript Compiler" (page 105) for more information about compiling Dart code into its JavaScript equivalent.

The Client's Dart Code

Dart code (client/chat-client.dart) provides the client's logic, using the DOM to interact with UI elements. For example, the client's Dart code uses the DOM to find the text area where the client displays messages.

Finding DOM Elements

The client app uses dart:html's top-level `query()` method to get the client's UI elements from the DOM.

```
// In client/chat-client.dart:
import 'dart:html';
//...
TextAreaElement chatElem = query('#chat-display');
InputElement usernameElem = query('#chat-username');
InputElement messageElem = query('#chat-message');
```

The `query()` method uses a selector string that identifies an element in the DOM. See "Finding elements" (page 68) for more about selectors.

Wrapping DOM Elements

Rather than always dealing with DOM APIs, the chat client wraps the elements in Dart objects:

```
chatWindow = new ChatWindow(chatElem);
usernameInput = new UsernameInput(usernameElem);
messageInput = new MessageInput(messageElem);
```

ChatWindow, UsernameInput, and MessageInput are custom classes that extend another custom class called View. These Views effectively separate the DOM manipulation from the application logic.

Because Dart has real classes and inheritance, it's simple to express the relationship that ChatWindow is-a View. Here's the complete code for UsernameInput:

```
class UsernameInput extends View<InputElement> {
  UsernameInput(InputElement elem) : super(elem);

  bind() { // Called by the View constructor.
    elem.on.change.add((e) => _onUsernameChange());
  }

  _onUsernameChange() {
    if (!elem.value.isEmpty()) {
      messageInput.enable();
    } else {
      messageInput.disable();
    }
  }

  String get username => elem.value;
}
```

To get the string that's in the chat-username field, the client app uses the `username` getter of a UsernameInput object. For example:

```
chatWindow.displayMessage(message, usernameInput.username);
```

Notice how the code uses generics (View<InputElement>) to specify what kind of element the View class can encapsulate. In the preceding example, the UsernameInput wraps an InputElement. Expressing this gives tools information that they can use to identify bugs or improve code completion.

Wrapping elements is a technique you can use as you develop a simple app that might evolve into a larger app. As the app grows, you might change it to use a real model-view-controller (MVC) architecture.

We expect the Dart project to provide an MVC-type framework for clients.

Updating DOM Elements

The `bind()` method sets up the event handlers, which bind events from the DOM to logic in the Dart objects. For example, in UsernameInput, the `_onUsernameChange()` method is called any time the text in the input element changes.

To display messages in the chat window, the ChatWindow class adds the message to the text node of the text area.

```
class ChatWindow extends View<TextAreaElement> {
  ChatWindow(TextAreaElement elem) : super(elem);

  displayMessage(String msg, String from) {
    _display('$from: $msg\n');
  }

  displayNotice(String notice) {
    _display('[system]: $notice\n');
  }

  _display(String str) {
    elem.addText(str);
  }
}
```

In both examples, the View objects expose an application-specific API—for example, `displayMessage()` or `_onUsernameChange()`—and encapsulate the manipulation of DOM elements.

Encoding and Decoding Messages

The dart:json library encodes and decodes JSON-formatted messages. JSON is an easy way to provide string message data to WebSockets. Using JSON also gives a bit of structure to the messages and leaves the door open to creating more detailed messages in the future.

The `stringify()` method converts a Dart object to a JSON encoded string, and the `parse()` method converts a JSON string back into a Dart object. Here's the JSON-related code from the chat client:

```
import 'dart:json';

var encoded = JSON.stringify({'f': from, 'm': message});
Map message = JSON.parse(encodedMessage);
```

See "dart:json - Encoding and Decoding Objects" (page 84) for more information.

Communicating with WebSockets

The custom class ChatConnection takes care of the chat client's WebSocket communication. First it connects to the WebSocket by calling the WebSocket constructor with the argument `'ws://127.0.0.1:1337/ws'`. Then ChatConnection adds event handlers for open, close, error, and message events, using the WebSocketEvents object it gets from `webSocket.on`. For example, here's the code that responds to message events:

```
webSocket.on.message.add((MessageEvent e) {
  print('received message ${e.data}');
  _receivedEncodedMessage(e.data);
});
```

The `_receivedEncodedMessage()` method just parses the JSON data and displays it in the status area.

```
_receivedEncodedMessage(String encodedMessage) {
  Map message = JSON.parse(encodedMessage);
  if (message['f'] != null) {
    chatWindow.displayMessage(message['m'], message['f']);
  }
}
```

To send a message on the WebSocket connection, `_sendEncodedMessage()` ensures the WebSocket connection is ready and then sends the JSON encoded message.

```
// In the ChatConnection class:
send(String from, String message) {
  var encoded = JSON.stringify({'f': from, 'm': message});
  _sendEncodedMessage(encoded);
}

_sendEncodedMessage(String encodedMessage) {
  if (webSocket != null && webSocket.readyState == WebSocket.OPEN) {
    webSocket.send(encodedMessage);
  } else {
    print('WebSocket not connected, message $encodedMessage not sent');
  }
}
```

In the event of a connection problem, the client code attempts to reconnect to the WebSocket server. The following code takes advantage of Dart's nested functions, nesting the `scheduleReconnect()` function inside of `_init()`. Dart's lexical scoping ensures that `scheduleReconnect()` can see variables from `_init()`.

```
_init([int retrySeconds = 2]) {
  bool encounteredError = false;
  chatWindow.displayNotice('Connecting to Web socket');
  webSocket = new WebSocket(url);
```

```
  scheduleReconnect() {
    chatWindow.displayNotice('socket closed, retrying in $retrySeconds seconds');
    if (!encounteredError) {
      window.setTimeout(() => _init(retrySeconds*2), 1000*retrySeconds);
    }
    encounteredError = true;
  }
  //...
  webSocket.on.close.add((e) => scheduleReconnect());
  webSocket.on.error.add((e) => scheduleReconnect());
```

The reconnect logic uses `setTimeout()` to schedule a retry using an exponential backoff algorithm.

The Server's Code

The chat-server.dart file contains most of the code used in the chat server. It is responsible for serving static files and managing WebSocket connections. The chat server also logs the chat messages to a file.

Serving Static Files

The chat server uses dart:io's HttpServer to implement a web server. The default request handler is configured to serve static files from a specific directory on the file system.

```
runServer(String basePath, int port) {
  HttpServer server = new HttpServer();
  server.defaultRequestHandler = new StaticFileHandler(basePath).onRequest;
  //...
  server.onError = (error) => print(error);
  server.listen('127.0.0.1', 1337);
  print('listening for connections on $port');
}

main() {
  var script = new File(new Options().script);
  var directory = script.directorySync();
  runServer('${directory.path}/client', 1337);
}
```

The StaticFileHandler first gets the file contents using File and InputStream, and then sends the contents using OutputStream.

Because I/O can cause delays, due to variable network or disk bandwidth conditions, the chat server uses asynchronous I/O to handle HTTP requests while still being responsive to other requests. Each I/O request returns a Future, allowing the server to continue executing without waiting for the I/O to complete.

For example, in the following snippet the `exists()` method returns a Future. When the Future completes (with a value of true if the file exists, or false if it doesn't), the function specified to `then()` executes.

```
// Respond to HTTP requests for static files.
onRequest(HttpRequest request, HttpResponse response) {
  //...
  file.exists().then((found) {
    if (found) {
      // ...Respond with the file's contents...
    } else {
      // ...Send a 404 response...
    }
  });
```

See "Asynchronous Programming" (page 64) for more information about using Future, and "Files and Directories" (page 80) for details on file and directory I/O.

Managing WebSocket Connections

In addition to serving static files, the chat server manages WebSocket connections, routing chat messages between clients. The dart:io WebSocketHandler class accepts HTTP connections, converts them into WebSocket connections, and then passes them to ChatHandler.

```
runServer(String basePath, int port) {
  //...
  WebSocketHandler wsHandler = new WebSocketHandler();
  wsHandler.onOpen = new ChatHandler(basePath).onOpen;
}
```

ChatHandler is a custom class that takes care of all WebSocket communication for the chat server. Here is its implementation.

```
class ChatHandler {
  Set<WebSocketConnection> connections;
  //...
  onOpen(WebSocketConnection conn) {
    connections.add(conn);

    conn.onClosed = (int status, String reason) {
      connections.remove(conn);
    };

    conn.onMessage = (message) {
      connections.forEach((connection) {
        if (conn != connection) {
          //...
          connection.send(message));
        }
      });
    };
```

```
      conn.onError = (e) {
        connections.remove(conn);
      };
    }
  }
```

When a client connects, the server adds the client's WebSocket connection to a collection. When the client disconnects (either through an error or on purpose), the server removes that client's connection from the collection. When a new message arrives, the server sends the message to all connected clients except the original source.

Logging Messages to a File

The chat server logs data to a file, client/chat-log.txt, using a custom library implemented in file-logger.dart. This library uses an isolate to handle file I/O without tying up the root isolate. Here's the code that creates and starts this isolate:

```
SendPort _loggingPort = spawnFunction(startLogging);
```

The value returned by dart:isolate's `spawnFunction()` is a SendPort object. Because isolates share no data, messages sent to ports are the only way for the root isolate to communicate with the spawned isolate.

The argument to `spawnFunction()` points to the `startLogging()` function, which implements the logging isolate. The logic for the logging isolate is simple: the first message specifies the log file location, and subsequent messages provide data to write to the log file.

```
startLogging() {
  print('started logger');
  File logFile;
  OutputStream out;
  port.receive((msg, replyTo) {
    if (logFile == null) {
      print('Opening file $msg');
      logFile = new File(msg);
      out = logFile.openOutputStream(FileMode.APPEND);
    } else {
      time('write to file', () {
        out.writeString('${new Date.now()} : $msg\n');
      });
    }
  });
}
```

In the preceding code, the port property used by `startLogging()` refers to a ReceivePort provided by dart:isolate. The port is how this isolate gets data from the root isolate. If this isolate needed to send messages back to the root isolate, it could use the replyTo argument (a SendPort) to do so.

Recall that in the root isolate, the _loggingPort variable holds a SendPort that the root isolate uses to send messages to the logging isolate. Every time the chat server calls the `log()` method, the root isolate sends the log data:

```
void log(String message) {
  _loggingPort.send(message);
}
```

See "dart:isolate - Concurrency with Isolates" (page 76) for more information about using isolates.

What Next?

You've seen how the Dart Chat sample uses both server-side and client-side Dart code to implement a web app. Here are some other samples you might want to look at:

- Solar, which simulates the solar system with animations in a canvas, using `requestAnimationFrame()`.
- Spirodraw, a fun, interactive tool to build colorful works of art.

Finally, please visit our website and join the discussion. We look forward to hearing from you!

- Dart website
- Dart discussion group
- Dart questions on Stack Overflow

About the Authors

Kathy is a technical writer who's worked on docs for Chrome and other developer APIs at Google since 2006. Before that, she worked at Sun, NeXT, and HP. Back when the Web was young, she wrote the first doc to help developers write Java applets. She also co-created *The Java Tutorial* and maintained it for a very long time.

Seth is a Developer Advocate with the Chrome team. He is a conference organizer (Aloha on Rails, New Game) and author (*Expert Spring MVC* [Apress]), helped publish Angry Birds for the Web, and is a big fan of HTML5 and the modern Web.