Dart

IN ACTION

Chris Buckett

FOREWORD BY Seth Ladd

6$0 3/(&+$37(5

Dart in Action

by Chris Buckett

Chapter 1

Copyright 2013 Manning Publications

brief contents

PART 1 INTRODUCING DART 1

1 ■ Hello Dart 3
2 ■ "Hello World" with Dart tools 24
3 ■ Building and testing your own Dart app 40

PART 2 CORE DART.. 69

4 ■ Functional first-class functions and closures 71
5 ■ Understanding libraries and privacy 94
6 ■ Constructing classes and interfaces 119
7 ■ Extending classes and interfaces 138
8 ■ Collections of richer classes 158
9 ■ Asynchronous programming with callbacks and futures 183

PART 3 CLIENT-SIDE DART APPS...209

10 ■ Building a Dart web app 211
11 ■ Navigating offline data 237
12 ■ Communicating with other systems and languages 258

PART 4 SERVER-SIDE DART 281

13 ■ Server interaction with files and HTTP 283
14 ■ Sending, syncing, and storing data 308
15 ■ Concurrency with isolates 331

Part 1

Introducing Dart

Dart is a great language for developing web apps. In chapter 1, you'll get an overview of why Dart was created and how Dart solves some of the problems experienced by many developers coming to web development. You'll discover some of the features the language offers and see why single-page web applications are a good architecture for building apps in Dart.

In chapter 2, you'll start to come to grips with the rich tool ecosystem that comes with Dart. Dart is more than a language—it's an entire development toolset, including an IDE, a custom developer browser for testing and debugging, and a Dart to JavaScript converter.

In chapter 3, you'll build a simple Dart app, learning how to create a browser-based, single-page web app. Through this example application, you'll be introduced to the language, including Dart's classes, functions, and variables. By the end of the chapter, you'll have a Dart project with a functioning user interface and accompanying unit tests, and you'll be ready to start learning about the core Dart language in Part 2.

1 Hello Dart

This chapter covers

- Basics of the Dart development platform
- A look at the Dart language
- Tools for building Dart applications

Dart is an exciting language that raises the possibility of building complex web applications more quickly and accurately than ever before. In this chapter, you'll find out how the Dart language and its tool ecosystem fit together, you'll discover some of the key features of the Dart language, and you'll see how you can you use Dart to begin building single-page web applications.

1.1 What is Dart?

Dart is an open source, structured programming language for creating complex, browser-based web applications. You can run applications created in Dart either by using a browser that directly supports Dart code or by compiling your Dart code to JavaScript. Dart has a familiar syntax, and it's class-based, optionally typed, and single-threaded. It has a concurrency model called *isolates* that allows parallel execution, which we discuss in chapter 15. In addition to running Dart code in web browsers and converting it to JavaScript, you can also run Dart code on the command line, hosted

in the Dart virtual machine, allowing both the client and the server parts of your apps to be coded in the same language.

The language syntax is very similar to Java, C#, and JavaScript. One of the primary goals for Dart was that the language seem familiar. This is a tiny Dart script, comprising a single function called `main`:

```
main() {                              <-- Single entry-point function main()
                                          executes when script is fully loaded
  var d = "Dart";                     <-- Optional typing
                                          (no type specified)
  String w = "World";                 <-- Type annotation
                                          (String type specified)
  print("Hello ${d} ${w}");           <-- Uses string interpolation to output "Hello
                                          Dart World" to browser console or stdout
}
```

This script can be embedded in an HTML page's `<script type="application/dart">` tags and run in the Dartium browser (a Dart developer edition of Google's Chrome web browser). You can convert it to JavaScript using the dart2js tool to run it in all modern browsers, or run the script directly from a server-side command line using the Dart Virtual Machine (Dart VM) executable.

There's more to Dart than just the language, though. Figure 1.1 shows the ecosystem of tools, which includes multiple runtime environments, language and editor tools, and comprehensive libraries—all designed to improve the developer's workflow when building complex web applications.

In addition to a great tool ecosystem that helps you build applications, Dart is designed to seem familiar, whether you're coming from a server-side, Java and C# world, or a client-side, JavaScript or ActionScript mindset.

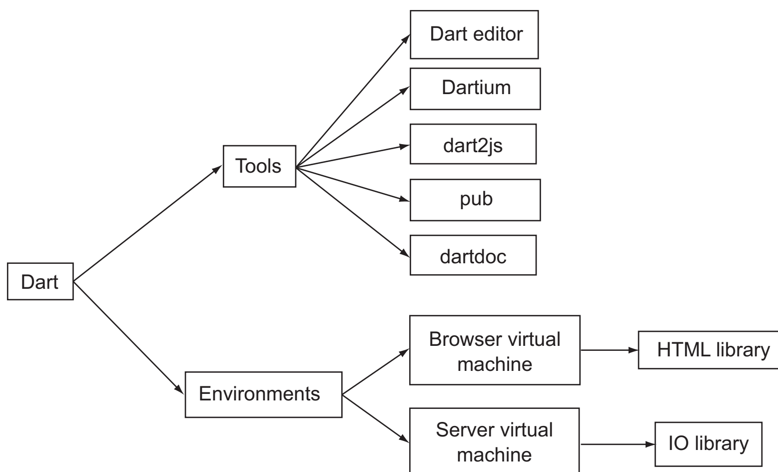


Figure 1.1 **Dart is more than just the language. The Dart project has an entire ecosystem.**

A key tool for Dart developers is Dartium, which lets you write or edit Dart code and see it running by loading the file and refreshing the browser. When Dartium is combined with the Dart Editor, you get the additional benefit of round-trip debugging.

1.1.1 A familiar syntax to help language adoption

One of the key design decisions was that Dart should be familiar to both JavaScript and Java/C# developers. This design helps developers who are new to Dart pick up the language quickly. If you're familiar with these other languages, you'll be able to read and understand the intent of Dart code without much trouble.

Java and C# developers are generally comfortable with type systems, classes, inheritance, and other such concepts. JavaScript developers, on the other hand, range from UI designers who copy and paste code to add interactivity to a web page (and have never used a type) to seasoned JavaScript programmers who understand closures and prototypical inheritance. To help with this developer diversity, Dart has an *optional typing* feature, which allows developers to specify absolutely no types (by using the `var` keyword, as in JavaScript), or use type annotations everywhere (such as `String`, `int`, `Object`), or use any mixture of the two approaches.

By using type information in your code, you provide documentation about your intent, which can be beneficial to automated tools and fellow developers alike. A typical workflow when building a Dart application is to build up the type information progressively as the code takes shape. Adding or removing type information doesn't affect how code runs, but it does let the virtual machine validate your code more effectively. This allows Dart's type system to bridge the gap between JavaScript's dynamic type system and Java's and C#'s static type system.

Table 1.1 provides some comparisons among Dart, Java, and JavaScript.

Table 1.1 High-level feature comparison among Dart, Java, and JavaScript

Feature	Dart	Java	JavaScript
Type system	Optional, dynamic	Strong, static	Weak, dynamic
First-class citizen functions	Yes	Can simulate with anonymous functions	Yes
Closures	Yes	Yes, with anonymous classes	Yes
Classes	Yes, single inheritance	Yes, single inheritance	Prototypical
Interfaces	Yes, multiple interfaces	Yes, multiple interfaces	No
Concurrency	Yes, with isolates	Yes, with threads	Yes, with HTML5 web workers

Dart is a general-purpose language, and like JavaScript or Java you can use it to build many different types of application. Dart really shines, though, when you're building complex web applications.

1.1.2 Single-page application architecture

The single-page applications Google Mail, Google Instant Search, and Google Maps are typical of the type of web application that Dart was designed to build. The source code for the entire application (or at least all the use cases for a major portion of the application) is loaded by a single web page. This source code, running in the browser, is responsible for building a UI and requesting data from the server to populate that UI, as shown in figure 1.2.

Single-page applications use a fast client-side virtual machine to move processing from the server to the client. This allows your server to serve more requests, because the processing involved in building the layout is moved onto the client. By using Dart's HTML libraries to incorporate modern HTML5 browser-storage and -caching technologies, applications can also cache data in the browser to improve application performance further or even allow users to work offline.

Each Dart script has a single entry-point function called `main()` that is the first function executed by the Dart VM. Thus you can rely on all code that defines an application when the `main` function is called; you can't define and execute a function within running code as you can with JavaScript—there is no `eval()` or other monkey-patching of executing code. This single feature helps you write Dart applications that fit the single-page application architecture, because you can be sure your code will execute as a single, known unit of code. The Dart VM uses this feature to improve application start-up time, using heap snapshots to load apps much more quickly than the equivalent JavaScript application.

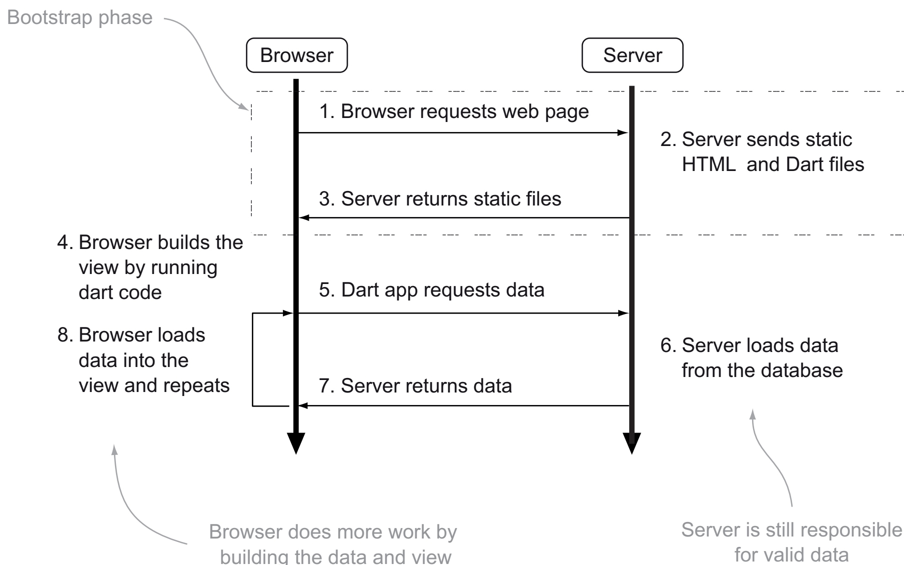


Figure 1.2 A single-page application runs in the browser, only requesting data from the server.

Remember

- Dart is a language for web development and has a familiar syntax.
- Dart's tool ecosystem provides greater productivity than equivalent dynamic languages.
- Dart's optional type system bridges the gap between JavaScript's dynamic typing and Java's static typing.
- Type annotations can greatly aid the development process among teams of developers by allowing tools to validate source code.
- Dart is ideal for developing single-page web applications.

Now that you've been introduced to Dart as a development platform, it's time to get hands-on with some of the key features of the Dart language.

1.2 A look at the Dart language

Dart is a fully featured, modern language. It has its roots in Smalltalk and is influenced by many other languages including Java, C#, and JavaScript. This section provides a grounding in some of the core concepts and highlights several complex pieces of the language that the book covers in detail.

Dart is an evolving language

At the time of writing, the Dart language is at a transition point between the experimental "technical preview" phase and a release that Google calls Milestone 1. Milestone 1 isn't version 1 but a line in the sand to allow features such as extended libraries surrounding the core language to be developed and enhanced. The Dart platform is intended to be a fully featured "batteries included" development environment, containing everything you need to build complex web applications. And Google, along with members of the Dart community, is now focused on building these libraries.

Milestone 1 also provides a neat baseline to enable you to start building applications, knowing that the breaking changes to the language syntax will be infrequent. Changes to the surrounding libraries, however, are likely, and the Dart Editor contains a helpful Clean-up tool that you can use to apply language and core library changes to your code.

1.2.1 String interpolation

Strings are used in many places throughout web applications. Dart provides a number of ways for you to convert expressions into strings, either via the `toString()` function that's built into the base `Object` class or by using string interpolation.

String interpolation uses the `$` character or `${ }` expression within single or double quotes. When you want to convert an expression to a string, you use the variable name with the `$` prefix, such as `$name`. If you want to use an expression that needs to be evaluated, such as a calculation or method call, include the curly braces:

```
"The answer is ${5 + 10}"
```

You can create multiline strings by using three double quotes; and you can write string literals (which ignore the `$` evaluation) by prefixing the string with an `r` character, such as `r'literal string'`. There is no `+` concatenator to join two strings together. You must use string interpolation such as `$forename $surname` or, if they're known string values, place them next to each other. For example,

```
var title = "Dart " "in " "Action";
```

produces a single string variable containing `"Dart in Action"`.

The following listing shows the things you can do with strings using Dart's built-in `print` function, which outputs to standard output, server-side, or the browser debug console when run in a browser.

Listing 1.1 String interpolation in Dart

```
void main() {

  var h = "Hello";
  final w = "World";
  print('$h $w');

  print(r'$h $w');

  var helloWorld = "Hello " "World";
  print(helloWorld);

  print("${helloWorld.toUpperCase()}");
  print("The answer is ${5 + 10}");

  var multiline = """
<div id='greeting'>
  "Hello World"
</div>""";
  print(multiline);

  var o = new Object();
  print(o.toString());
  print("$o");

}
```

- $ evaluates simple variables
- r prefix outputs literal string without interpolation
- Adjacent string constants are concatenated
- Evaluated expressions need to be within braces ${ }
- Multiline strings ignore first line break following """
- Multiline strings can contain both single and double quotes
- String interpolation automatically calls toString() function

The output from this listing is

```
Hello World
$h $w
Hello World
HELLO WORLD
The answer is 15
<div id='greeting'>
  "Hello World"
</div>
Instance of 'Object'
Instance of 'Object'
```

You'll use string interpolation and the `print` function a lot when experimenting with Dart, logging variables to help with debugging, and inserting values into HTML snippets.

1.2.2 Optional types in action

One of the key differences between JavaScript and Dart is that Dart has the concept of types baked into the language. Fortunately, by using Dart's *option typing*, you can get the benefit of strong typing through type annotations where you use them.

Optional type annotations are used in variable declarations, for function parameter definitions and return types, and in class definitions. The following snippet shows four ways of declaring the string variable `message`. The first two have no type annotations, and the second two provide the `String` type annotation, indicating to developers and tools that you intend a string value to be used in the variable:

```
var messageA;
var messageB = "Hello Dart";

String messageC;
String messageD = "Hello Dart";
```

No type annotations provided (messageA, messageB)

Type annotations provided (messageC, messageD)

In the previous snippet, two of the variable declarations initialize the value of `message` at the time it's declared. If the value won't change after declaration, then you should use the `final` keyword, as shown here:

```
final messageE = "Hello Dart";
final String messageF = "Hello Dart";
```

Uses final with no type annotation (messageE)

Uses final with type annotation (messageF)

We'll cover the `final` keyword in more detail later in the book.

As an example of how you can benefit from using optional typing, consider the following block of code, which has a `trueIfNull()` function that takes two parameters and returns `true` if both are null (and `false` if not). This code has no type annotations at present, but we'll explain how you can use type annotations to show intent:

```
trueIfNull(a, b) {
  return a == null && b == null;
}

main() {
  final nums = trueIfNull(1,2);
  final strings = trueIfNull("Hello ", null);
  print("$nums");
  print("$strings");
}
```

Function takes two values (trueIfNull)

Stores "false" in dynamic variable nums

Stores "true" in dynamic variable strings

Outputs variables nums and strings to console

No type annotations are provided in the snippet, which means that when reading this code, you have no idea about the developer's intent. The `trueIfNull(a,b)` function could mean that `trueIfNull(a,b)` should take two `int` types and return a `bool` (`true`/`false` value), but the developer could have intended something else—for example, to return the string `"true"` instead of a `bool`. Dart's optional typing allows the developer to provide documentation in the form of type information about the parameters and return types:

```
bool trueIfNull(int a, int b) {
  return a == null && b == null;
}

main() {
  final bool nums = trueIfNull(1,2);
  final bool strings = trueIfNull("Hello ", null);
  print("$nums");
  print("$strings");
}
```

Adds return type and parameter types

Adds type information about variable declarations

NOTE The previous example contains a `bool` type. In Dart, unlike in JavaScript, there is a single false value: that of the keyword `false` itself. Zero and null don't evaluate to false.

Adding this type information doesn't change the running of the Dart application, but it provides useful documentation that tools and the VM can use to validate the code and find type errors. Dart can be said to be *documentary typed* because the code will run the same without the types. Any type information provided can help the tools during static analysis (in the Editor or from the command line as part of a continuous build system) and at runtime. Future developers who may maintain your code will also thank you.

TIP Use specific types (for example, `String`, `List`, and `int`) where doing so adds documentary value, such as for function parameters, return types, and public class members; but use `var` or `final` without type annotations where it doesn't, such as inside function bodies. The Dart style guide available at www.dartlang.org recommends this approach. You should get used to seeing a mix of code like this, because it's the way Dart was intended to be written.

Optional typing is core to many of Dart's mechanisms and appears throughout the book, where the syntax is different enough from Java and JavaScript to warrant explanation. Functions are covered specifically in chapter 4.

1.2.3 Traditional class-based structure

Dart uses classes and interfaces in a traditional and unsurprising object-oriented way. It supports single inheritance and multiple interfaces. If you aren't familiar with class-based OO programming, it would probably be useful to read about the subject at one of the many resources on the web. At this point, it's enough to point out that Dart's OO model is similar to Java/C# and not similar to JavaScript. We'll look at classes and their features in greater depth in chapters 6 and 7.

All Dart classes inherit by default from the `Object` class. They can have public and private members, and a useful getter and setter syntax lets you use fields interchangeably with properties without affecting users of the class. The next listing shows a quick example of a class.

Listing 1.2 A simple class in Dart

```
class Greeter {
  var greeting;
  var _name;

  sayHello() {
    return "$greeting ${this.name}";
  }

  get name => _name;
  set name(value) => _name = value;
}

main() {
  var greeter = new Greeter();
  greeter.greeting = "Hello ";
  greeter.name = "World";
  print(greeter.sayHello());
}
```

- class keyword defines new class
- Public property
- Private property denoted by _
- Public method
- Uses String interpolation
- Getter and setter with shorthand syntax
- new keyword creates new instance of Greeter
- Assigns values to fields and setters with same syntax

This simple class contains a lot of functionality. Private members are indicated by prefixing the name with the _ (underscore) character. This convention is part of the Dart language, with the benefit that you can instantly tell when you're accessing a method or property in private scope when you're reading code.

The getter and setter syntax is also useful because you can use the fields of a class the same way you use getters and setters. Thus a class designer can expose the property (such as `greeting`, in listing 1.2) and later change it to use a getter and setter (such as in `name` in the example) without needing to change the calling code.

The `this` keyword, which causes a lot of misunderstanding in the JavaScript world, is also used in a traditional OO fashion. It refers to the specific instance of the class itself and not the owner of the class (as in JavaScript) at any given point in time.

> **Classes are optional**
>
> Unlike in Java and C#, classes are optional in Dart. You can write functions that exist in top-level scope without being part of a class. In other words, you don't need to declare a class in order to declare a function. If you find that you're writing classes that contain utility methods, you probably don't need a class. Instead, you can use Dart's top-level functions.

1.2.4 Implied interface definitions

Dart has interfaces just like Java and C#, but in Dart, you use the class structure to define an interface. This works on the basis that all classes define an implicit interface on their public members. Listing 1.3 defines a class called `Welcomer` and a top-level `sayHello()` function that expects a `Welcomer` instance. In addition to using the `extends` keyword to implement inheritance of the sort found in Java and C#, you can

also use the interface defined on each class by using the `implements` keyword. The `Greeter` class `implements` the public methods of `Welcomer`, which allows it to be used in place of a `Welcomer` instance. This lets you concentrate on programming against a class's interface rather than the specific implementation.

Listing 1.3 Every class has an implicit interface

```
class Welcomer {
  printGreeting() => print("Hello ${name}");
  var name;
}

class Greeter implements Welcomer {
  printGreeting () => print("Greetings ${name}");
  var name;
}

void sayHello(Welcomer welcomer) {
  welcomer.printGreeting();
}

main() {
  var welcomer = new Welcomer();
  welcomer.name = "Tom";
  sayHello(welcomer);

  var greeter = new Greeter();
  greeter.name = "Tom";
  sayHello(greeter);
}
```

Welcome class can be created and inherited from ...

... but also has an implied interface that Greeter implements.

Expects Welcomer argument

Because Greeter implements a Welcomer interface, it can be used in place of Welcomer.

This ability to implement a class that doesn't have an explicit interface is a powerful feature of Dart. It makes mocking classes or providing your own custom implementation of a class relatively straightforward; you don't need to inherit explicitly from a shared base class.

1.2.5 Factory constructors to provide default implementations

In addition to having a constructor syntax similar to Java and C#, Dart has the concept of *factory constructors*. This lets the class designer define a base class to use as an interface, and supply a factory constructor that provides a default concrete instance. This is especially useful when you intend a single implementation of an interface to be used under most circumstances.

Listing 1.4 shows an `IGreetable` class that has a factory constructor to return an instance of a `Greeter` class. The `Greeter` class implements the interface defined on `IGreetable` and lets users of the interface use the default `Greeter` implementation without knowing they're getting an implementation of `Greeter`. Thus the class designer can change the specific implementation without users of the `IGreetable` interface being aware of the change.

Listing 1.4 Factory constructors for default implementations

```
abstract class IGreetable {
  String sayHello(String name);

  factory IGreetable() {
    return new Greeter();
  }
}

class Greeter implements IGreetable {
  sayHello(name) {
    return "Hello $name";
  }
}

void main() {
  IGreetable myGreetable = new IGreetable();
  var message = myGreetable.sayHello("Dart");
  print(message);
}
```

Defines interface

Provides method that must be implemented

Factory constructor returns instance of Greeter

Greeter implements IGreetable interface

Creates instance of IGreetable, which returns Greeter implementation

Uses Greeter implementation

Because of this ability, it's important to note that a number of the core classes are interfaces—for example, `String` and `int`. These have specific implementation classes that are provided using factory constructors. I cover classes, interfaces, and their interaction with the optional type system at length in part 2 of the book.

1.2.6 Libraries and scope

Dart has the ability to break source code files into logical structures. It's possible to write an entire Dart application in a single .dart file, but doing so doesn't make for great code navigation or organization. To address this issue, Dart has libraries baked into the language. A *library* in Dart is a collection of source code files that *could* have been a single file but have been split up to aid human interaction with the code.

I mentioned earlier that classes are optional in Dart. This is so because functions can live in the top-level scope of a library. In Dart, a library is one or more .dart files that you group together in a logical fashion; each file can contain zero or more classes and zero or more top-level functions. A Dart library can also import other Dart libraries that its own code uses.

A library is defined using the `library` keyword, imports other libraries using `import`, and refers to other source files using `part`, as shown in the following listing.

Listing 1.5 Libraries and source files

```
library "my_library";

import "../lib/my_other_library.dart";

part "greeter.dart";
part "leaver.dart";

greetFunc() {
```

Declares that file is a library

Imports another library from a different folder

Includes other source files (containing Greeter class)

Defines function in top-level library scope

```
  var g = new Greeter();
  sayHello(g);
}
```

Uses class from greeter.dart file

Calls function in top-level scope of my_other_library

From the listing, you can see that it's possible to define a method in the top-level scope of a library—that is, without it being part of a class. Therefore, you need to define classes only when you need to instantiate an object, not when you just need to collect a group of related functions.

Libraries can pull a group of source files into the same scope. A library can be made up of any number of source files (including zero), and all the source files put together are equivalent to having a single library file containing all the separate files' code. As such, each source file can reference code that's in another source file, as long as both source files are part of the same library. Each source file can also reference code that's exposed by importing other libraries, as in the example `my_other_library.dart` in figure 1.3.

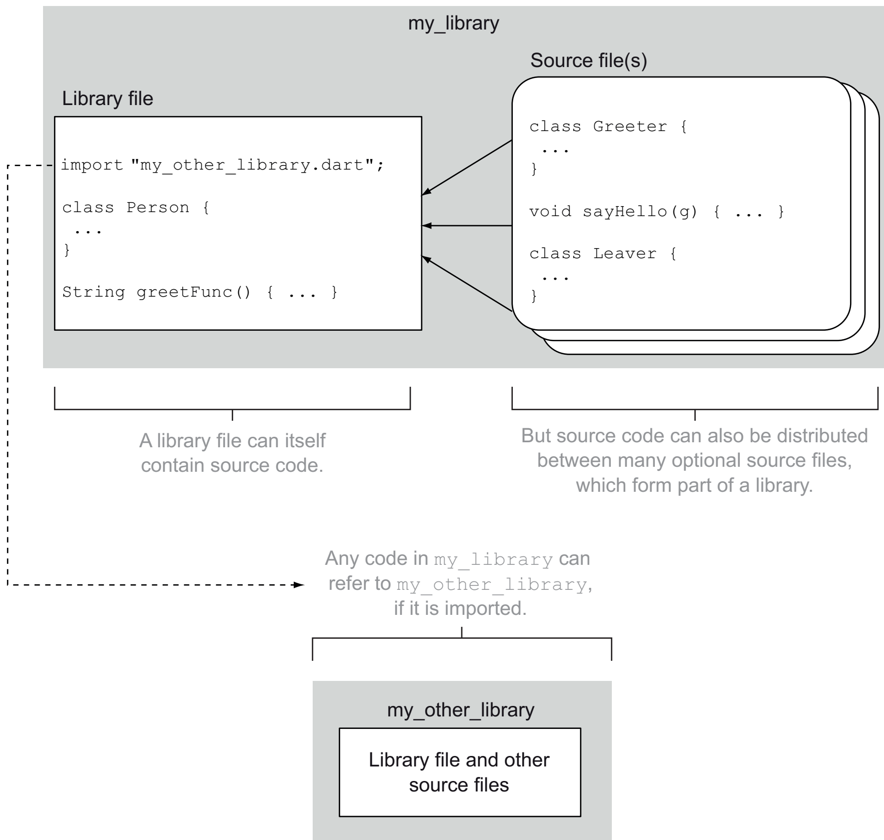


Figure 1.3 **my_library.dart is made from greeter.dart and leavers.dart and uses another library called my_other_library.dart (which in turn is constituted from various source files).**

To avoid naming conflicts, such as when a library that you're writing and a library that you're importing both contain a class called `Greeter`, you can apply a library prefix such as

```
import "../lib/my_other_library.dart" as other;
```

and then refer to classes in that library using the form

```
other.otherLibFunction("blah");
```

Thus it's possible to ensure that you name your classes and methods sensibly without having to worry about polluting a global namespace.

A library can form the entry point of your application, but if it does, it must have a `main()` function. You can have multiple libraries, each with its own `main()` function, but it's the `main` function in the library referred to in the `<script>` tag that's executed.

Any functions or classes in your library are made available to any importers of your library; that is, they're public. To stop importers of your libraries from using specific functions or classes, mark them as private.

CLASS AND LIBRARY PRIVACY

Although libraries and classes can be useful to modularize your application, good practice dictates that you keep the workings of your class or library private. Fortunately, Dart provides a simple method for making things private: prefix the name of a method, function, class, or property with an underscore (_). Doing so makes the item library private, or private within the scope of a library.

PRIVATE, BUT ONLY WITHIN A LIBRARY

In our ongoing example, if something is marked as private with the underscore, it means that if class `Greeter` and class `Leaver` are in the same library, they can access each other's private elements (similar to `package private` in Java). It also means that a property or function `_greeterPrivate()` is accessible from any other class in the same library. But when `Greeter` is imported via another library, it isn't visible to that other library, as shown in figure 1.4.

Private elements can include top-level (library) functions, classes, class fields, properties and methods (known as members), and class constructors. From within a library, privacy is ignored, such that any `part` file can access private elements in another `part` file if the `library` brings those source files into the same library. Users of the library can't access any private elements of that library (or private elements of classes within that library).

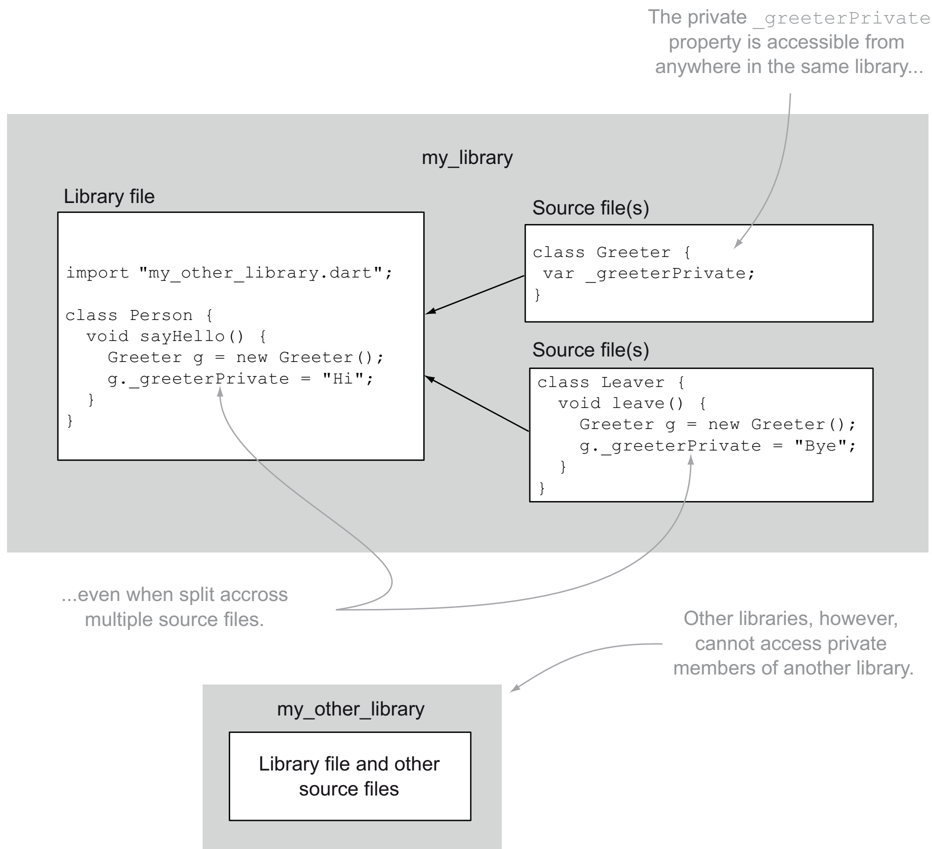


Figure 1.4 **Private elements such as fields, methods, library functions, and classes are private within a library. Privacy is indicated by an _ prefix. Users of the library can't access private elements.**

1.2.7 Functions as first-class objects

You can pass around functions in Dart as objects, in a manner similar to in JavaScript: you can pass a function as a parameter, store a function in a variable, and have anonymous (unnamed) functions to use as callbacks. The next listing gives an example of this feature in action.

Listing 1.6 Functions as first-class objects

```
String sayHello(name) => "Hello $name";        // ❶ Declares function using function shorthand

main() {
  var myFunc = sayHello;                       // ❷ Assigns function into variable

  print(myFunc("World"));                      // Calls function stored in variable

  var mySumFunc = (a,b) {                      // Defines anonymous function
    return a+b;
  };
```

```
  var c = mySumFunc(1,2);
  print(c);
}
```

Calls anonymous function

(1) is a single-line function that uses `=>` shorthand to return a value. The following two functions are identical:

```
String sayHello(name) {
  return "Hello $name";
}

String sayHello(name) => "Hello $name";
```

With a function defined, you can get a reference to it and store it in a variable (2). You can pass this around like any other value. Anonymous functions such as the one stored in the variable `mySumFunc` are often used in event-handler callbacks. It isn't uncommon to see a block of code like

```
myButton.on.click.add((event) {
  // do something
});
```

(with the anonymous function indicated in boldface).

1.2.8 Concurrency with isolates

Dart is a single-threaded language. Although this design may be at odds with current hardware technology, with more and more processors being available to applications, it means that Dart has a simple model to code against.

In Dart, the *isolate* (rather than the thread or process) is the unit of work. It has its own memory allocation (sharing memory between isolates isn't allowed), which helps with the provision of an isolated security model. Each isolate can pass messages to another isolate. When an isolate receives a message, which might be some data to process, an event handler can process that message in a way similar to how it would process an event from a user clicking a button. Within an isolate, when you pass a message, the receiving isolate gets a copy of the message sent from the sending isolate. Changes to the received data aren't reflected on the sending side; you need to send another message back.

In a web page, each separate script (containing a `main()` function) runs in its own isolate. You might have scripts for different parts of your application, such as one for a news feed, one for offline syncing, and so on. Dart code can spawn a new isolate from running code in a way similar to how Java or C# code can start a new thread. Where isolates differ from threads is that an isolate has its own memory. There's no way to share a variable between isolates—the only way to communicate between isolates is via message passing.

Isolates are also used for loading external code dynamically. You can provide code outside of your application's core code that can be loaded into its own memory-protected space and that will run independently of your app, communicating via message passing. This behavior is ideal for creating a plug-in architecture.

The Dart VM implementation may use multiple cores to run the isolates, if required. And when isolate code is converted to JavaScript, they become HTML5 web workers.

> **Remember**
>
> - Dart has optional (or documentary) typing.
> - Libraries help you break up source files and organize code.
> - Privacy is built into the language.
> - Functions are first-class and can exist without classes.
> - Dart understands concurrency using message-passing isolates.

Now that you've seen a high-level view of what Dart looks like, let's look at how you can use it to program the web.

1.3 Web programming with Dart

One of Dart's aims is to improve the life of developers. And because Dart is ultimately a programming language for the web, a significant amount of effort has gone into turning the browser DOM-manipulation API into something that's a joy to use. In JavaScript, accessing the browser DOM was a chore until jQuery was created, which made it feel natural to work with the browser. Similarly, the `dart:html` library was written to ease the writing of browser code in Dart.

1.3.1 dart:html: a cleaner DOM library for the browser

At the time of writing, no UI widget library is available for Dart. Although the Dart team has publicly stated that they expect Dart to be a "batteries included" language, the early public release of Dart means that they need to spend some time getting the language working perfectly before the focus moves to higher-level abstractions. But they have built what could be considered the equivalent of jQuery core, in the form of `dart:html`.

If you've used a framework like jQuery, then you'll be familiar with using CSS selectors to access DOM elements such as DIVs with `id="myDiv"` or all the `<p>` elements. The `dart:html` library makes this work easy. Rather than including a number of different calls to get elements, such as `getElementsById()` and `getElementsByName()`, as you would with native DOM APIs, `dart:html` has only two methods for selecting elements: `query()`, which returns a single element, and `queryAll()`, which returns a list of elements. And because the `dart:html` library uses Dart lists, you can use all the standard list functions such as `contains()` and `isEmpty()`, and array syntax such as `element.children[0]`. The following listing shows some interaction with the DOM via the `dart:html` library.

Listing 1.7 Interacting with the browser

```
import 'dart:html';                                  // Imports dart:html library

void main() {
  var button = new Element.tag("button");            // Creates new button element
```

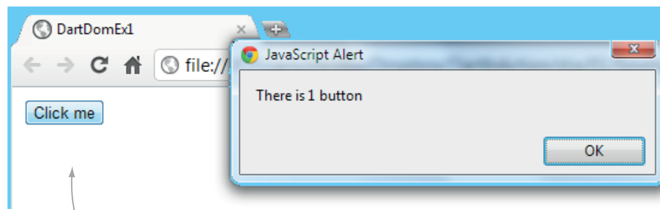
```
  button.text = "Click me";
  button.on.click.add((event) {
    List buttonList = queryAll("button");
    window.alert("There is ${buttonList.length} button");
  });

  document.body.children.add(button);
}
```

Adds anonymous function (in bold italic) as event handler to on.click event

Adds button to HTML body

The listing uses a named constructor to create a button. The button is given an event handler (using an anonymous function) and added to the document body. Running this example produces the output shown in figure 1.5.



Clicking the button triggers the event handler function, which was added with `button.on.click.add (...)`

Figure 1.5 The output from clicking the button in listing 1.7

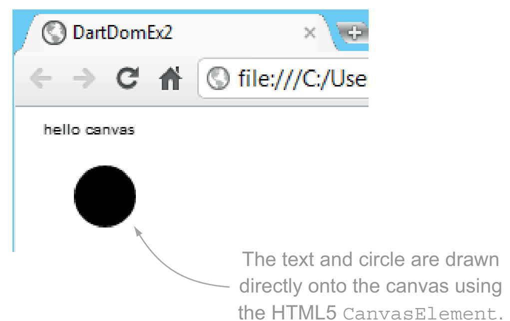
By interacting with the browser in this fashion, you can create complex UIs entirely in Dart code and CSS. We'll explore this in chapter 4.

> **TIP** When you're writing browser code, remember that the `print` function used in `print("Hello World")` sends output to the browser console, not to the page. You can access the browser console in Chrome under the Wrench > Tools > JavaScript Console menu option.

1.3.2 Dart and HTML5

Just as you can interact with browser elements directly, the `dart:html` library also exposes HTML5 elements such as the canvas, WebGL, device motion events, and geolocation information. The output shown in figure 1.6 is produced by the code in listing 1.8, which uses the HTML5 Canvas API.

The Dart code that draws this output to the canvas adds an HTML5 `<canvas>` tag to the browser DOM and then uses it to get a 2D drawing context. The Dart code then writes text and shapes onto the drawing context.



The text and circle are drawn directly onto the canvas using the HTML5 `CanvasElement`.

Figure 1.6 Drawing on the browser canvas

Listing 1.8 Drawing on the browser canvas

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

void main() {
  CanvasElement canvas = new Element.tag("canvas");      <-- Creates new CanvasElement
  canvas.height = 300;
  canvas.width = 300;
  document.body.children.add(canvas);      <-- Adds canvas to document body

  var ctx = canvas.getContext("2d");      <-- Gets drawing context from canvas

  ctx.fillText("hello canvas", 10, 10);      <-- Writes text

  ctx.beginPath();
  ctx.arc(50, 50, 20, 0, PI * 2, true);
  ctx.closePath();
  ctx.fill();      | Draws filled circle

}
```

By creating a `CanvasElement` and adding it to the page, you get an area that you can draw directly onto using the standard drawing methods such as `drawImage`, `fillText`, and `lineTo`. We'll look at this in more detail in chapter 10.

With the `dart:html` library, you have ready access to all the standard browser elements that you'd expect to code against. And because the DOM library, which forms part of the `dart:html` library is generated from the WebKit IDL (Interface Definition Language), you can be sure of getting access to the up-to-date browser functionality available in Dart.

Remember

- `dart:html` provides a Dart view of the browser DOM.
- HTML5 support is a core part of the Dart language.

Now that you have some knowledge of Dart running in the browser, it's time to look at the tools available to help you write Dart.

1.4 The Dart tool ecosystem

The Dart tools are considered a feature of the Dart platform and as such are undergoing development as rapid as that of the language. As developers, we tend to experience any particular language through the available tools (or lack thereof), and Google is putting a lot of effort into this area. The place to start is editing code.

1.4.1 The Dart Editor

Although you can use any text editor to write Dart code, you'll get the best experience when you use the Dart Editor. The Dart Editor is built using the Eclipse Rich Client Platform (RCP), a framework for building customized code editors. In the Dart Editor,

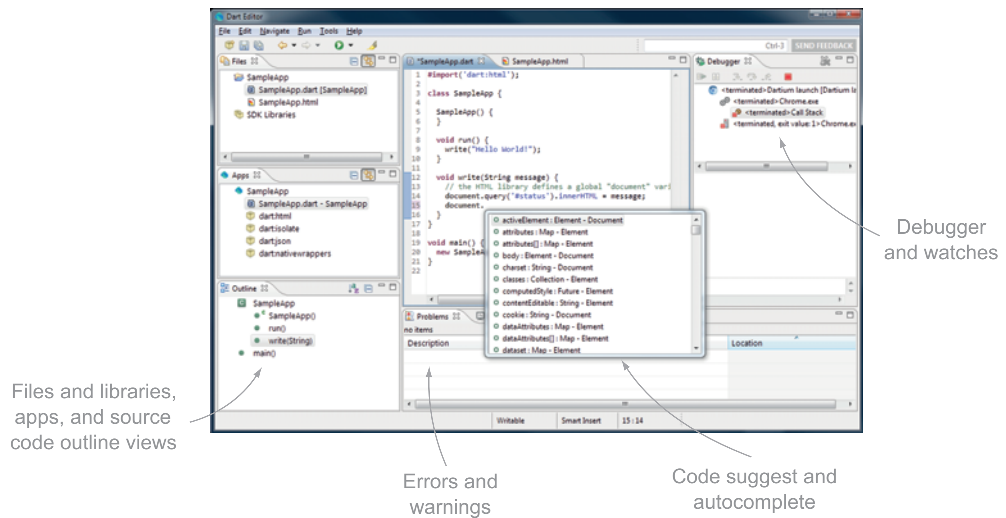


Figure 1.7 The Dart Editor, showing a simple browser application and the code-completion window

you get the usual features such as code completion, navigation, and code outlining, along with static analysis such as warnings and errors. The static-analysis tool is also available as a standalone command-line tool that you can use in your continuous-build system to provide early indication of errors in the code. Figure 1.7 shows a typical view of some of the features in the Dart Editor.

Using the Dart Editor, you can write code; and if that code is associated with an HTML page, you can convert the code into JavaScript and open it in a browser of your choice by using the dart2js tool. In the Dartium browser, which is Chrome with the Dart VM embedded, you can skip the conversion to JavaScript and execute the code directly in the browser. Dartium also communicates back to the Dart Editor to allow round-trip, step-by-step debugging.

If your code isn't associated with an HTML page, the editor will run the code as though it were executed from the command line, outputting to the `stdout` console.

1.4.2 Dart virtual machine

The Dart VM is the core of the Dart language. One use is as an executable on the command-line VM (which allows you to run Dart code on the console), such as to start up an HTTP server or run a simple script (equivalent to a batch file or shell script), or any other console-based use of Dart. Another use is to embed it in another application, such as Dartium.

1.4.3 Dartium

Dartium is a customized build of Chromium (the open source version of Google Chrome) with the Dart VM embedded in it. It recognizes the script type `application/`

`dart` and executes Dart code natively in the browser without requiring conversion to JavaScript. It includes the developer tools that are familiar to many web developers who build websites and web applications with Chrome. Coupled with the Dart Editor, it provides step-and-continue debugging: you can add breakpoints to the editor and then refresh your app in the Dartium browser; the Editor's debugger will stop on the correct breakpoint, allowing you to inspect variables and step through instructions.

The Dartium browser makes developing Dart as simple as developing JavaScript. A simple browser refresh is the only step you need to run your Dart code.

1.4.4 dart2js: the Dart-to-JavaScript converter

You use the dart2js tool to compile Dart into JavaScript, from within the Dart Editor or standalone on the command line. The dart2js tool compiles all the various libraries and source code files that make up a Dart application into a single JavaScript file. The code it outputs is fairly readable, although when you use Dartium to develop natively in Dart, you'll seldom need to read it.

dart2js also produces JavaScript source maps, which allow you to hook back from the output JavaScript to the original Dart code. This recent innovation is also used successfully in other languages that convert to JavaScript, such as CoffeeScript and Google Web Toolkit (GWT).

> **NOTE** dart2js is the third Dart-to-JavaScript converter. The first was dartc, and the second was a tool called frog. You may see these names in various older documents and blog posts; they're all tools to convert Dart to JavaScript.

1.4.5 Pub for package management

Package management is a key feature of any language, with Maven for Java, NuGet for .NET, and npm for node.js being common examples. Dart has its own package manager called pub. Pub lets library developers define package metadata in a pubspec file and publish their libraries in code repositories such as GitHub.

When you use a library, you can use the pub tool to download all the various libraries that your app requires, including versioned dependencies. We'll discuss this more and show an example of using pub in chapter 5 when we look at Dart's library structure.

> **Remember**
>
> - The Dart tool ecosystem forms a core part of the Dart project.
> - The Dart Editor provides rich tooling for developers.
> - Dartium makes developing in Dart as simple as a browser refresh.
> - Dart is designed to be converted to JavaScript.

1.5 Summary

At first glance, Dart might be seem like just another language. But when you take into account the entire Dart ecosystem, Dart represents an exciting prospect in the world of web development. With applications becoming more complex and requiring larger development teams, Dart and its associated tools and environments promise to provide some structure in the previously overly flexible world of JavaScript.

Single-page applications hosted in a browser (such as Google Plus) become more achievable with a language like Dart, because maintaining a large client-side code base becomes less fragile. Dart—with its ability to either run natively or be converted to JavaScript—coupled with HTML5 is an ideal solution for building web applications that don't need external plug-ins to provide features.

In the following chapters, you'll play with the Dart ecosystem, explore the core language, and use Dart to develop single-page web applications that target modern HTML5-capable web browsers. By the end of the book, you'll be developing Dart applications that run offline in the client, are served from a Dart file server, and connect to a Dart server to persist data in a database.

WEB DEVELOPMENT

Dart IN ACTION

Chris Buckett

Dart is a web programming language developed by Google. It has modern OO features, just like Java or C#, while keeping JavaScript's dynamic and functional characteristics. Dart applications are "transpiled" to JavaScript, and they run natively in Dart-enabled browsers. With production-quality libraries and tools, Dart operates on both the client and the server for a consistent development process.

Dart in Action introduces the Dart language and teaches you to use it in browser-based, desktop, and mobile applications. Not just a language tutorial, this book gets quickly into the nitty-gritty of using Dart. Most questions that pop up while you're reading are answered on the spot! OO newbies will appreciate the gentle pace in the early chapters. Later chapters take a test-first approach and encourage you to try Dart hands-on.

What's Inside

- Dart from the ground up
- Numerous code samples and diagrams
- Creating single-page web apps
- Transitioning from Java, C#, or JavaScript
- Running Dart in the browser and on the server

To benefit from this book you'll need experience with HTML and JavaScript—a Java or C# background is helpful but not required.

Chris Buckett builds enterprise-scale web applications. He runs Dartwatch.com and is an active contributor to the dartlang list.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/DartinAction

"Includes numerous examples of core language features as well as more advanced HTML5 features."
—From the Foreword by Seth Ladd, Developer Advocate, Google

"A compelling and captivating book about learning and working with Dart as an alternative to JavaScript."
—Glen Stokol
Oracle Corporation

"Puts the future of web apps in the palm your hand."
—Rokesh Jankie, QAFE, Inc.

"The perfect guide for a beautiful language."
—Willhelm Lehman
Websense, Inc.

ISBN 13: 978-1-617290-86-2
ISBN 10: 1-617290-86-6

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]