Dart Essentials

Design and build full-featured web and CLI apps using the powerful Dart language and its libraries and tools

Martin Sikora
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Dart'
- A synopsis of the book’s content
- More information on Dart Essentials

About the Author

Martin Sikora has been programming professionally since 2006 for companies such as Miton CZ and Symbio Digital in various languages, mostly PHP. Since 2012, he's been freelancing, working on projects in Python, PHP, Dart, Objective-C, and AngularJS. He's a Zend Certified Engineer and was a member of the winning team at Google Dart Hackathon 2012 in Prague.
Dart Essentials

Since the first public release of Dart in 2011, the language evolved a lot. Its first stable release in November 2013 marked the point when it was the time to start taking it seriously in the production environment. Dart 1.9+ might, at first sight, look like what JavaScript could be if it was designed for today's Web. It combines the best of many other languages while targeting both browser and server-side development.

Dart isn't the answer to every problem on the Web. There are situations where it seems like it's overly complicated to use Dart or it just isn't good at what you need, and you might be right. There are circumstances where it's better to use JavaScript.

This book tries to be objective. We won't try to convince you that from now on, you should use only Dart, and we'll talk about situations where Dart isn't the ideal solution. At the same time, we'll show you where Dart is great, how you can write well-structured code for both the browser and the server, and how easy it is to write understandable asynchronous apps with Dart 1.9.

Even though this book can't go into great detail about each topic, after reading this book, you should see for yourself that Dart makes sense and you should at least give it a try. It's not just the language but also the IDE, comfortable debugger, dependency management, runtime profiler, and more.

What This Book Covers

Chapter 1, Getting Started with Dart, jumpstarts Dart development right away while explaining the syntax and core features of Dart.

Chapter 2, Practical Dart, focuses on the most common tasks of client-side development, such as DOM manipulation, asynchronous programming, Ajax calls, and using existing JavaScript code in Dart and vice versa.

Chapter 3, The Power of HTML5 with Dart, specifically focuses on using HTML5 features in Dart while mentioning some noteworthy third-party libraries written for Dart.

Chapter 4, Developing a Mobile App with Dart, builds on the previous chapter with a quick explanation of CSS3 transformations and HTML5 features specific for mobile devices.
Chapter 5, *Web Components and polymer.dart*, goes step by step, showing each part of Web Components standard and how useful they are, even when used separately. Then, it looks at polymer.dart, which combines all parts of Web Components into a single framework.

Chapter 6, *AngularDart*, is a sneak peak of a superheroic framework for Dart made by Google with the same philosophy in mind as AngularJS.

Chapter 7, *Server-side Applications with Dart*, shows that apart from the browser environment, there's also a standalone Dart VM, which can run on a server. We'll take a very practical look at writing server-side scripts, including server configuration for Apache and nginx web servers.

Chapter 8, *Testing and Profiling the Dart Code*, states that just as with any other language, unit testing is a vital part of the development process. Dart also comes with a built-in tool called Observer to examine Dart VM's internals in runtime.

Chapter 9, *Writing Native Extensions for the Standalone Dart VM*, shows the full potential of Dart, by writing native extensions for the standalone Dart VM in C/C++ and then using them from Dart.
Getting Started with Dart

The first chapter will introduce us to the world of Dart. We'll learn:

• What is so interesting about Dart and why it's worth your time to give it a try
• Where and how to get the Dart SDK with its IDE called Dart Editor
• How to create the first app and see how to run and debug it in Dartium
• How to compile our app to JavaScript

What is Dart?

The Dart language was first unveiled at the GOTO conference in Aarhus in October 2011. Its primary goal, in the long run, is to replace JavaScript as the only language in browsers.

Although JavaScript is very easy to use for smaller apps, with the increasing complexity and the necessity to scale today's projects, it quickly becomes very hard to maintain. Frankly, JavaScript wasn't designed for this and writing larger apps is just a pain.

Dart was created as a brand new language with C-style syntax; it's object-oriented and class-based with a single inheritance model with mixins. It gives you many things that you've probably already used in other languages, such as abstract classes, encapsulation, reflection, exceptions, and so on. On the top of that, you can make use of optional static type checking.

Then, if you look a little deeper, you'll find things such as Future-Based API for all asynchronous calls, typedefs, isolates, streams, zones, dependency management, and even more out of the box. You'll probably find many of these things already familiar, but as you'll see, Dart uses a very easy-to-understand approach, which allows you to stay focused on writing your apps instead of dealing with the language itself.
In November 2013, Dart reached its first stable release, 1.0, and is still in active development.

On March 23, 2015, the Dart team released version 1.9, which significantly simplified working with asynchronous APIs and is considered the most important release since version 1.0.

At the end of April 2015, Google held the first Dart Summit revealing plans to use Dart as a language for cross-platform mobile development with their new runtime called Fletch.

Why choose Dart?
There are five main environments where you can run Dart:

- Dart compiled to JavaScript in practically any modern browser. The Dart SDK is shipped with Dart to a JavaScript compiler called dart2js, which takes your Dart code and compiles it into vanilla JavaScript. Right now, this is the most typical use case in a production environment.
- The Dartium web browser comes out of the box with the Dart SDK. It's a modified version of Chromium (basically, an open sourced Chrome) that contains Dart VM along with JavaScript V8. We'll use Dartium for easy debugging and to run the Dart code without compiling it to JavaScript.
- The standalone Dart VM allows you to run Dart code as a CLI script just like any other scripting language. The dart2js compiler itself is written in Dart.
- Google Cloud Platform, since the introduction of Google's Managed VMs in November 2014, it also supports a hosting environment for server-side code written in Dart.
- Fletch is an experimental highly concurrent Dart runtime, which will be able to run on both desktop and mobile platforms. It is scheduled for its official release at the end of 2015.

Dart VM is able to run your Dart code significantly more effectively than JavaScript V8. You can see current benchmarks at www.dartlang.org/performance/. Note that the dart2js compiler is doing pretty good even though the compilation process brings some additional overhead.

After the first public release of Dart, Google claimed they were planning to implement Dart VM right into Chrome as an alternative to JavaScript. However, on March 25, 2015, the Dart team released a blog post stating that after collecting reactions from their internal teams, they've decided to stop efforts to integrate Dart VM with Chrome and rather focus on improving integration with JavaScript.
Installing the Dart SDK

Let's start with obtaining the Dart SDK, which already contains all we need. This includes Dart Editor, the standalone Dart VM, and Dartium browser.

We can download everything in a single package right from Dart's home page at https://www.dartlang.org/, which detects your platform automatically for you and gives you the correct package for download.

Dart Editor

Dart Editor is built on the Eclipse platform, so you might find its look and feel already familiar.

![Dart Editor screenshot]

We're going to spend a lot of time in this editor, so feel free to take a look at it and try the example projects. This editor also contains a debugger that works out of the box with Dartium, so we don't need to configure anything.
Although the browser that comes with the Dart SDK package is actually called Chromium, we call it Dartium because it runs Dart VM inside. Even in Dart Editor, they call it Dartium, so don't be confused when you hit Run in Dartium and it opens the Chromium browser.

There's also a Dart plugin for IDEs developed by JetBrains, specifically, WebStorm, PHPStorm, and IntelliJ IDEA.

We're going to use Dart Editor in this book. However, the Dart team announced that they're planning to move to JetBrains IDEs and abandon Dart Editor probably in late 2015.

**Writing a greeting for all Dartsans**

Our first Dart app will randomly generate five colors in the `<ul>` element; let's enter a name into the `<input>` field and greet you with a selected color inside `<h1>`.

The final working app with some CSS will look like this:

```html
<body>
  <ul id="color-select">
    <li class="selected" style="background-color: rgb(45, 192, 143);">!</li>
    <li style="background-color: rgb(45, 192, 143);">!</li>
    <li style="background-color: rgb(220, 152, 164);">!</li>
    <li style="background-color: rgb(150, 62, 88);">!</li>
    <li style="background-color: rgb(91, 90, 118);">!</li>
  </ul>
  <input type="text" id="name">
  <div id="output">
    "Hello, "
  </div>
  <script type="application/dart" src="/main.dart"></script>
  <script type="application/dart" src="/packages/browser/dart.js"></script>
</body>
```

We'll set off by creating a new project by clicking on Create an application in the Welcome window or by going to **File | New Project**. There are a few templates for the most common use cases. We'll go with Uber Simple Web Application because we need just the most basic app structure right now.
Chapter 1

Our project should look like this:

For us, the most important files are `pubspec.yaml`, `index.html`, and `main.dart`. We can take a look at them one by one.

**pubspec.yaml**
This is a file that defines our project and its dependencies. By default, it contains only very basic information and one dependency called `browser`, which we'll use in `index.html`. If you're using Dart Editor, you can add more dependencies right in the editor's GUI, and you don't need to modify the file as text. Later in this chapter, we'll add more statements that control, for example, the `dart2js` compiler. For now, we can leave it as it is:

```yaml
name: 'my_first_dart_app'
version: 0.0.1
description: An absolute bare-bones web app.
environment:
  sdk: '>=1.0.0 <2.0.0'
dependencies:
  browser: any
```

Note that dependencies in Dart projects don't necessarily need to contain any Dart code. For example, `browser` contains only two JavaScript files.
Getting Started with Dart

Downloading the example code
You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

When you modify pubspec.yaml, Dart Editor downloads new dependencies automatically for you.

We used any to specify the version for the browser package, which means that the newest available version will be used. There are more ways to define allowed versions; for a more detailed description, refer to https://www.dartlang.org/tools/pub/dependencies.html. We’ll use this option to set specific versions when working with polymer.dart and AngularDart in Chapter 5, Web Components and polymer.dart, and Chapter 6, AngularDart.

index.html
This is going to be just a simple HTML page:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>my_first_dart_app</title>
  <link rel="stylesheet" href="styles/main.css">
</head>

<body>
  <ul id="color-select"></ul>
  <input type="text" id="name">
  <h1 id="output"></h1>

  <script type="application/dart" src="main.dart"></script>
  <script data-pub-inline src="packages/browser/dart.js"></script>
</body>
</html>
```
Look at the last two `<script>` tags. The first one links the `main.dart` file, which is an entry point for our app. No matter how many files your Dart project has, you always link just the one that contains the `main()` function, as we'll see in a moment.

The `browser` package contains a script called `dart.js` that you'll probably use in all the Dart web projects you'll make. When you compile the Dart code to JavaScript, it creates a new file called `main.dart.js` with all your Dart code compiled to JavaScript. The `dart.js` script automatically checks whether your browser supports Dart natively and if it doesn't, it replaces `main.dart` with `dart.main.js`. Therefore, you can develop, test, and deploy projects in both Dart and JavaScript without even touching the HTML code. The `data-pub-inline` attribute tells the compiler to handle this element in a special way. We'll talk about this later in this chapter.

In this file, we created three elements (`<ul>`, `<h1>`, and `<input>`) that will be controlled from Dart.

The `data-pub-inline` attribute tells the compiler to handle this element in a special way. We'll talk about this later in this chapter.

We're omitting the CSS file here and in most of the book as well, unless there's something particularly interesting and related to the topic. You can download all the source code for this book from the Packt Publishing website.

### The `main()` function

The real fun starts here. The entry point to the app is the top-level `main()` function and as Dart is a class-based language, we'll create a class called `GreetingsManager` that will update the text and its color.

We can jump right into the code to get a quick glimpse of what Dart code looks like. Try to read the code and guess what you think it does. I believe that even without any knowledge of Dart, you'll be able to tell how it works.

```dart
// web/main.dart
import 'dart:html';
import 'dart:math';

class GreetingsManager {
  Random _rnd; // Random number generator.
  HtmlElement h1 = querySelector('#output');

  GreetingsManager() {
    _rnd = new Random();
  }
}
```
// Generate a random color.
String getRandomColor() {
  // Randomly generate numbers as hex strings with padding.
  return _rnd.nextInt(256).toRadixString(16).padLeft(2, '0') +
    _rnd.nextInt(256).toRadixString(16).padLeft(2, '0') +
    _rnd.nextInt(256).toRadixString(16).padLeft(2, '0');
}

// Generate a list of strings where each item represents one
// color. [List<T>.generate()] is a named constructor that
// calls the callback function n-times.
List<String> generateColors(int total) {
  return new List<String>.generate(total, (int i) =>
    getRandomColor());
}

void setTextColor(String color) {
  SpanElement span = h1.querySelector('span');
  if (span != null) {
    span.style.color = color;
  }
}

void setText(String name) {
  // Override its inner HTML.
  h1.innerHTML = name.trim().isEmpty
    ? ""
    : "Hello, <span>$name</span>!";
}

void main() {
  var gm = new GreetingsManager();

  // Target container for our colors.
  UListElement ul = querySelector('#color-select');

  // Iterate all colors and create <li> element for each of them.
  gm.generateColors(5).forEach((String color) {
    LIElement elm = new LIElement();
    // Set the background color.
    elm.style.backgroundColor = "#${color}";
  });
}
// Bind a listener to the onClick event.
elm.onClickListener.listen((e) {
  gm.setTextColor(elm.style.backgroundColor);
  ul.querySelectorAll('li').classes.remove('selected');
  elm.classes.add('selected');
});
// Add HTML element to the <ul>.
ul.append(elm);

InputElement nameInput = querySelector('#name');
// Bind a listener to the onKeyUp event.
nameInput.onKeyUp.listen((Event e) {
  String name = (e.target as InputElement).value;
  // print() outputs a variable as a string to
  // the environment's standard output.
  print(name);

  gm.setText(name);
  LIElement selected = ul.querySelector('li.selected');
  if (selected != null) {
    gm.setTextColor(selected.style.backgroundColor);
  }
});

There are a couple of important things to pay attention to in more detail.
The code starts with import statements. These tell Dart to import (as you've probably guessed) another file or a package. Starting with dart:, it means that this is a built-in package that's shipped with the Dart SDK. Later, we'll also use package:, which is a third-party dependency, and at the end of the book, we'll meet dart-ext:, which is a native extension of the Dart VM. Of course, we'll use import to import files from our own projects as well.

All web apps will probably import the dart:html library because it makes top-level variables, document and window, and methods, such as querySelector() or querySelectorAll(), available.

Then, we declared a GreetingsManager class. If we didn't write a constructor for it, Dart would use the so-called implicit constructor by default. There's also a named constructor that we'll meet later.
All types in Dart are optional, but we're going to use them a lot in this book. It's not only easier to read; it also helps you spot possible errors and in some situations improves performance when compiled to JavaScript. If you don't care what type a variable is, you can declare it as `var` like in JavaScript, and the Dart static check won't bother you with it. There's also a special type `dynamic`, which is used underneath every time you don't specify a variable type, but in most situations, you're just fine with declaring variables with `var`. The `dynamic` keyword makes more sense when used as a generic keyword for `List` and `Map` classes, as we'll see later.

Every method in Dart has a return type, although you can use `dynamic` and `void` as well (omitting return type stands for `void`). Void means that this method doesn't return any value. Note that `void` doesn't have the same meaning as `null`. Null means zero or an undefined pointer, which is a valid value, while `void` means `nothing` in this context.

Collections such as lists are defined in Dart's API as `List<E>`. This means that the `List` class is generic and you can tell it what type of objects it may contain. In our example, we defined `List<String>`, which tells the type checker that all items in this collection will be instances of `String`. This notation of generics is very common in Java and C++, but as Dart types are optional, it actually doesn't restrict you from adding instances of other classes to the list, as you might expect. Using generics properly in Dart is a matter of a good programming style because it helps you and other developers understand your intentions. By the way, this is another situation where you can use the `dynamic` type if your list can contain any objects. As types are optional in Dart, declaring `List<dynamic>` is equal to not using the `<E>` notation at all. We'll see this in use later.

Notice the way we access HTML element properties and how we can change their CSS style with `elm.style.backgroundColor`. Adding, removing, or toggling the classes of an element is very easy because the `classes` property is an instance of `CssClassSet`, which has many useful methods, and we can use `elm.classes.add('selected')`, for example. With Dart, most of the time, you don't need to access element attributes directly.

To remove element's classes, we can use `querySelectorAll('li').classes.remove('selected')`, where `querySelectorAll()` returns a collection of elements and `classes.remove('selected')` on each of them. This approach is well known from jQuery, and it saves you a lot of writing the same code over and over again.
Then, we have the main() function, which is an entry point to our app. Dart VM parses your code before running it, so it doesn't matter where in the file you put it (it still has to be a top-level function). There, we call the GreetingsManager.generateColors() method and chain it with the forEach() method. All iterable collections implement the forEach() method, which calls a callback for each item in the collection. Creating an anonymous function has two possible formats. A short one-liner with just one expression, which we used in generateColors(), is as follows:

```dart
(int i) => getRandomColor()
```

This takes one parameter, calls getRandomColor(), and returns its result. This notation is equivalent to the second and is probably a more common format:

```dart
(int i) {
  return getRandomColor();
}
```

There is also another way we could iterate the entire collection:

```dart
for (String str in gm.generateColors()) {
  /* ... */
}
```

Listening to events is done via Dart streams, which is basically a way of handling asynchronous calls. For the most part, we can use them just like events in JavaScript. In our app, we're listening to the onKeyUp and onClick events. We "listen" to them by calling the listen() method that takes a callback function as an argument.

Dart lets you use type casting in a similar way to C/C++ with the variable as type notation (where as is a keyword). This is useful when the static checker doesn't know what type of object is stored in a variable but you know what you're expecting. We used it like (e.target as InputElement).value because we know that e.target is going to be of the InputElement type but e.target is a general dynamic property that doesn't have the value property itself. Of course, we could omit the typecast completely and just ignore the warning shown by type checker, but that's not a very good practice.

The last interesting thing is string interpolation. We can concatenate String objects in Dart with a plus sign +, but this tends to be confusing when used too much. Therefore, we can insert variables right into the string and leave Dart to do the work for us. In our app, we used it like this:

"Hello, <span>$name</span>!"
Getting Started with Dart

The \$variable notations are replaced with a string representation of their variables. Interpolation can be used for expressions as well with $\{expr\}$, for example, $\{42. toString()\}$.

Running and debugging code in Dartium

Our code is done for now, so we can run the app in the browser. First, we'll test it in Dartium because it's able to run the native Dart code.

You can right-click on index.html and select Run in Dartium, or you can click on the white arrow in the green circle icon in the toolbar. This opens the Dartium browser and you should see the same page with five random colors just like what's shown at the beginning of this chapter. Open Developer Tools and see whether the browser really uses our main.dart file.

Debugging Dart scripts is very easy because we can use exactly the same tools used for debugging JavaScript. With Developer Tools, we can only debug web apps and not console apps.
Another way to debug both web and console apps is right in Dart Editor. Double-click on any line number to place a breakpoint and Dart VM will pause when it reaches that line.

Compiling Dart to JavaScript

From a practical point of view, Dart would be useless if we couldn't run it in today's browsers. This is why the Dart SDK comes with Dart to JavaScript compiler called dart2js. You can run it right in Dart Editor; in the top menu bar, navigate to Tools | Pub Build or right-click on index.html and select Run as JavaScript. This launches the compilation process and outputs some info:

```plaintext
--- 3:12:49 AM Running pub build on ./my_first_dart_app ... ---
Loading source assets...
Building my_first_dart_app...
[Info from Dart2JS]:
  Compiling my_first_dart_app|web/main.dart...
[Info from Dart2JS]:
  Took 0:00:07.294964 to compile my_first_dart_app|web/main.dart.
Built 224 files to "build".
```
As you can see, the compiler had to process 224 files in total and generated one large main.dart.js file, which we already mentioned earlier in this chapter. The compiler created a new directory named build and put there everything you need to run the app in both Dart and JavaScript.

You can run the compiler in CLI by navigating to your project's directory and running:

```
$ pub build
```

This command fetches all the dependencies, compiles your code with dart2js, and eventually processes it with transformers.

A very obvious question is how big the generated JavaScript file is. The Dart compiler removes parts of the code that your app isn't using and in Dart SDK 1.9, the final script is 290 KB. That's not bad but especially for mobile connections, it's still quite a lot. Luckily for us, we can tell dart2js to minify the final JavaScript by adding a new statement at the end of pubspec.yaml (you have to open the file as a text file or switch to the Source tab at the bottom of Dart Editor's window):

```
transformers:
  - $dart2js:
      minify: true
```

When we run the compilation again, it will generate a 125 KB file. That's much better; keep in mind that this also includes all our project's code. For comparison, jQuery 2.1 that doesn't contain legacy code for older browsers and without our app code has 84 KB. With gzip compression enabled on your server, the difference is even smaller: 37 KB for dart2js versus 30 KB for jQuery. With the latest Dart Editor 1.9, you can create minimized version right from the Tools menu; however, setting it specifically in pubspec.yaml is sometimes necessary when using Dart's pub tool in CLI (more about this in the next chapter).
There's still one more thing to optimize. Our index.html includes JavaScript called dart.js, which we've already talked about. The template that we used has a special attribute, data-pub-inline:

```
<script data-pub-inline src="packages/browser/dart.js"></script>
```

By default, it does nothing. Let's add a new dependency to our project called script_inliner and then update pubspec.yaml again with:

```
transformers:
  - $dart2js:
    minify: true
  - script_inliner
```

Then, run Pub Build again; script_inliner processes HTML files and inlines JavaScripts marked as data-pub-inline.

### The Dart language tour

We've already gone through many concepts of the Dart language when explaining the code, but there are still some aspects that we should look at in greater detail.

The Dart team maintains an official style guide for Dart at [https://www.dartlang.org/articles/style-guide](https://www.dartlang.org/articles/style-guide).

### The static checker

Dart VM can run in production or checked mode. The checked mode performs runtime type checking and runs all assert(condition) checks. Asserts are a way to quickly test whether a condition is true, and if it's not, then stop the script execution. Runtime type checks can, for example, detect when you declare a variable int and then try to assign it a String object.

This is useful when developing or testing your app, but it introduces some overhead, and for this reason, Dart VM runs in production mode by default and simply ignores all these checks.

Dart Editor launches Dartium with enabled checked mode, but you can switch to production mode in the top menu window by navigating to Run | Manage Launches and by unchecking Run in checked mode for the configuration that runs in Dartium. For the standalone Dart VM, you can enable the checked mode with the -c parameter (for example, in command line, dart -c main-cli.dart).
Variables

We’ve already seen how to declare variables and we also saw that all types in Dart are optional. Apart from this, we can define variables as const or final.

The final keyword lets you assign a value to a variable only once. This behavior is slightly different when assigning objects because final only locks a variable's values and when you're assigning objects, you're working with pointers (their memory addresses). In other words, you're assigning a memory address to a variable and it doesn't care what the internal state of the object is. It can change in runtime. The final keyword only prevents you from assigning a different value to the same variable. For example, you can declare an instance of List<String> as final and add items dynamically during the app's lifetime.

The const keyword lets you assign a value to a variable at compile time. For this reason, you can't make a variable const if its value is dependent on an expression that isn't determinable at compile time. The same restrictions are applied when instantiating objects where all their properties have to be compile time determinable too.

All uninitialized variables have the null value.

Built-in types

Everything in Dart is an object; however, some types are treated in a special way:

- **Numbers**: Built-in int and double types are both subclasses of the num abstract class. All integers and floating point numbers are stored as int or double, respectively.
- **Strings**: All strings in Dart are stored as String objects. Both single and double quotes to enclose a string are allowed. Classes can implement the toString() method, which should return a textual representation of the inner state of an object.
- **Booleans**: These are true/false values. Note that in JavaScript, a value is considered to be true if it's a nonzero length string, nonempty array, any value different from 0, and so on. But in Dart, only the Boolean value true is treated as true, nothing else.
- **Lists**: All lists (or arrays) are instances of List class. You can use short notations to create lists just like in JavaScript with this:
  ```dart
  List<T> list = [1, 2, 3];
  ```
• **Maps:** Key-value storage in Dart is represented by the `Map` class. You can use short notation with this as well:

```dart
Map<K,V> map = {
  'key1' => 'value',
  'key2' => 123
};
```

You can get a value associated to a key with a square bracket notation, such as `map['key']`. In contrast to JavaScript, the `Map` class in Dart has a public property, `length`, which represents the total number of key-value pairs in the map.

### Functions and parameters

Defining functions is nothing new, and we've already seen this in the preceding sections. Dart extends function definitions with optional positional and named parameters, both well known from Python.

After the required parameters, you can specify any number of optional parameters.

Named parameters are mostly useful when you have a function with many parameters, and in most use cases, you need to use only a few of them. You can define named parameters as a map with curly brackets, `{}`:

```dart
void myFunc(required, {bool opt1: true, String opt2: "hello"}) {
  /* ... */
}
```

Another method is to use optional positional parameters. You can define positional parameters with square brackets, `[]`:

```dart
void myFunc2(required, [bool named1, String named2]) {
  /* ... */
}
```

One function definition **can't combine both** named and positional optional parameters. Some examples of calling both types could be:

```dart
myFunc(true, opt2: "Hello, World!");
myFunc2(false, true, "I'm Dart");
```
Class properties

When defining classes, there are no public, protected or private keywords like in PHP or Java. Instead, every property that starts with _ is private. Everything else is public. Dart also generates default getters and setters for each property, but you can override them or create your custom properties:

```dart
Class Square {
  num height;
  num width;

  num get size => height * width;
  set size(int value) {
    width = height = sqrt(value);
  }
}
```

You can later use size just like any other public property:

```dart
var cls = new Square();
cls.size = 49;
```

This updates both width and height.

Class inheritance and abstract classes

Dart is an object-oriented language with a mixin-based inheritance model. This means that every class has exactly one superclass but can implement multiple classes as interfaces and use multiple class bodies (mixins). Every class is at least a subclass of the Object class.

Abstract classes can't be instantiated directly and often contain abstract methods.

Any class in Dart can be treated as an interface for another class, which might be a little unusual:

```dart
abstract class MyAbsClass {
  String hello();
}

class MyInterface {
  String hello2() => "hello!";
}

class MyInterface2 {
  String anotherHello() => "hello!";
}
```
class MyClass extends MyAbsClass implements MyInterface, MyInterface2
{
    String hello() => "Ahoy";
    String hello2() {
        /* ... */
    }
    String anotherHello() {
        /* ... */
    }
}

Note that the hello() abstract method in MyAbsClass doesn't need a keyword in front of it. It just doesn't have any body. The MyClass class implements two classes as its interfaces and automatically takes all their methods as abstract and requires you to implement them. This isn't the same as inheritance because it completely ignores methods' bodies and expects you to write them by yourself. There's also keyword super that refers to the superclass.

In later chapters, we'll explain and use mixins as well.

**Constructors**

There are two types of constructors, generative and named. Generative constructors are the same constructors as in any other language. Their name equals the class name:

```plaintext
class MyClassWithConst {
    num height;
    num width;

    MyClassWithConst(num w, num h) {
        width = w;
        height = h;
    }
}
```

Assigning default values to object properties is very common, so there's a shorter way to write the preceding constructor with just the following:

```plaintext
MyClassWithConst(this.width, this.height);
```
Since there's no constructor overloading (like in Java or C++), Dart offers named constructors:

```dart
class MyClassWithConst {
  /* ... */
  MyClassWithConst(this.width, this.height);

  MyClassWithConst.withZeros() {
    this.width = 0;
    this.height = 0;
  }
}
```

Note that constructors aren't inherited from superclasses and you don't even have to define any constructor and Dart will use a default one for you (just the class name with no parameters).

**Exceptions**

Dart supports throwing exceptions when something goes wrong. The logic behind this is just like in any other language, so it probably doesn't require any further explanation:

```dart
throw new Exception("It's broken");
```

Catching exceptions is based on typical `try`, `catch`, and `final` statements. You can catch only specific exceptions or make general `catch` statements for all exceptions:

```dart
try {
  brokenFunction();
} on MyOwnException {
  itsNotThatBad();
} catch (e) {
  itsPrettyBad();
} finally {
  cleanup();
}
```

Custom defined exceptions have to inherit from the `Exception` class.

**Using static types**

It might look like it's better to always specify variable types everywhere. The rule of thumb here is to use types when it's unclear what the variable type is, including return types. For example, look at the following code:

```dart
GreetingsManager gm = new GreetingsManager();
```
Instead of writing the class name twice, we use the following:

```dart
var gm = new GreetingsManager();
```

The static analyzer knows that the `gm` variable is assigned to a `GreetingsManager` class, so it's fine to use just `var`. This also applies when iterating collections:

```dart
List<String> colors = ['red', 'green', 'blue'];
for (var color in colors) {
}
```

We don't need to declare `String color` because this is obvious from the `List<String>` declaration.

The same approach is recommended when using `void` and `dynamic`. If there's no good reason to specify them, just omit them.

**Summary**

This is probably enough for the first chapter. There're still topics to be covered and we'll explain more Dart concepts in later chapters when we come across them.

You can see that Dart uses a lot of ideas from other languages and that it requires very little knowledge to be able to start using it.

In the next chapter, we'll write another app that uses a lot of the stuff that we learned now. We'll also take a look at using existing JavaScript code in Dart and vice versa, which is a very interesting and practical topic for every Dart learner. For easier migration from other languages to Dart, you can take a look at the list of synonyms in Dart and other languages at [https://www.dartlang.org/docs/synonyms/](https://www.dartlang.org/docs/synonyms/).
Where to buy this book

You can buy Dart Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.