

1

Dart – A Modern Web Programming Language

In this chapter we will investigate:

- What Dart is all about
- Why it is a major step forward in the web programming language arena

We will get started with the Dart platform and have a look at its tools. Soon enough we will be programming and taking a dive in a simple functional todo list program, so that you realize how familiar it all is.

What is Dart?

Dart is a new general and open source programming language with a vibrant community developed by Google Inc. and its official website is <https://www.dartlang.org>. It was first announced as a public preview on October 10, 2011. Dart v1.0, the first production release, came out on November 14, 2013, guaranteeing a stable platform upon which production-ready apps can be built. World class language designers and developers are involved in this project, namely, *Lars Bak* and *Kasper Lund* (best known from their V8 JavaScript engine embedded in the Chrome browser, which revolutionized performance in the JavaScript world) and *Gilad Bracha* (a language theorist known from the development of the Strongtalk and Newspeak languages and from the Java specification). Judged by the huge amount of resources and the number of teams working on it, it is clear that Google is very serious about making Dart a success.



Take your time to familiarize yourself with the site [dartlang.org](https://www.dartlang.org). It contains a wealth of information, code examples, presentations, and so on to supplement this book, and we will often reference it.

Dart looks instantly familiar to the majority of today's programmers coming from a Java, C#, or JavaScript/ ActionScript background; you will feel at ease with Dart. However, this does not mean it is only a copy of what already exists; it takes the best features of the statically typed "Java-C#" world and combines these with features more commonly found in dynamic languages such as JavaScript, Python, and Ruby. On the nimble, dynamic side Dart allows rapid prototyping, evolving into a more structured development familiar to business app developers when application requirements become more complex.

Its main emphasis lies on building complex (if necessary), high-performance, and scalable-rich client apps for the modern web. By modern web we mean it can execute in any browser on any kind of (client) device, including tablets and smart phones, taking advantage of all the features of HTML5, and is ported to the ARM-architecture and the Android platform. Dart is designed with performance in mind, by the people who developed V8. Because the Dart team at Google believes web components will be the foundation for the next evolution of web development, Dart comes out of the box with a **web component** library (web components are pieces of web code containing HTML and Dart or JavaScript that you can re-use in different pages and projects, in other words it is a reliable infrastructure of widgets).

But Dart can also run independently on servers. Because Dart clients and servers can communicate through web sockets (a persistent connection that allows both parties to start sending data at any time), it is in fact an end-to-end solution. It is perfect on the frontend for developing web components with all the necessary application logic, nicely integrated with HTML5 and the browser document model (DOM). On the backend server side, it can be used to develop web services, for example, to access databases, or cloud solutions in Google App Engine or other cloud infrastructures.

Moreover, it is ready to be used in the multicore world (remember, even your cell phone is multicore nowadays) because a Dart program can divide its work amongst any number of separate processes, called **isolates**, an actor-based concurrency model as in Erlang.

Dart is a perfect fit for HTML5

To appreciate this fully we have to take a look at the history of client-side web development.

A very short history of web programming

A web application is always a dialog between the client's browser requesting a page and the server responding with processing and delivering the page and its resources (such as pictures and other media). In the technology of the early dynamic web (the 90s of the previous century, extending even until today), the server performed most of the work: compiling a page, executing the code, fetching data from a data store, inserting the data in the page templates, and in the end producing a stream of HTML and JavaScript that was delivered to the browser. The client digested this stream, rendering the HTML into a browser screen while executing some JavaScript, so processing on the client side was minimal. The whole range of applications using Perl, Python, Ruby, PHP, JSP (Java Server Pages), and ASP.NET follows this principle. It is obvious that the heavy server loads impact negatively the number of clients that could be served, as well as the response time in these applications. This mismatch is even clearer when we realize that the power of the client platforms (with their multicore processors and large memories) is heavily underutilized in this model.

The plugin model, in which the browser started specialized software (such as the Adobe Flash Player) for handling all kinds of media and other resources, partly tipped the balance to the client side. Following this trend, a whole range of technologies for developing **Rich Internet applications (RIA)** were developed that executed application code in the browser at the client side instead, ranging from Java applets around 1995 and Microsoft Active X Objects, culminating in the Adobe Flex and Microsoft Silverlight frameworks. While they have their merits, it is clear that they are more like a parasite in the browser; for example, a virtual machine that executes code, such as ActionScript or C#, that is alien to the browser environment.

Dart empowers the web client

Empowering the client is the way to go, but this should better be done with software technology intimately linked to the browser itself: HTML and JavaScript. In order to eliminate the need for alien plugins, the power of HTML needs to be enlarged, and this is precisely what is achieved with **HTML5**, for example, with its `<audio>` and `<video>` tags. **JavaScript** is the ubiquitous language of the Web and it can, as with Dart, request/send data from/to the server without blocking the user experience through technologies such as Ajax. But flexible and dynamic as JavaScript may be, today it is also often called the assembly language for the web.

JavaScript is not the way to go for complex apps

Why is this? JavaScript was from the beginning not designed to be a robust programming language, despite its name that suggests an affiliation with Java. It was designed to be a simple interpreted language that could be used by nonprofessional programmers and that would be complemented by Java for more serious work. But Java went away to prosper on the server, and JavaScript (JS for short) invaded the browser space. Today JS is being used to develop big and complex web applications, with server components such as Node.js, far beyond the original purpose of this language. Most people who have worked on a large client-side web application written entirely in JS will sooner or later come to the conclusion that its use in these applications is overstretched and the language was not meant to build that kind of software.

Understanding program structure is crucial in large, complex applications: this makes code maintenance, navigating in code, debugging, and refactoring easier. But unfortunately JS code is hard to reason about because there is almost no declarative syntax and it is very hard to detect dependencies between different scripts that can appear in one web page. JavaScript is also very permissive: almost anything (spelling mistakes, wrong types, and so on) is tolerated, making it hard to find errors and unpredictable results. Furthermore, JS allows you to change the way built-in core objects function, a practice often called monkey patching (for a reason!). Would you trust a language in which the following statement is true in its entirety and all of its comparisons?

```
10.0 == '10' == new Boolean(true) == '1'
```

Because of this sometimes undefined nature of JS, its performance is often very unpredictable, so building high-performance web apps in it is tricky.

Google, GWT, and Dart

Google is the web firm par excellence: its revenue model is entirely based on its massive web applications, such as Gmail (some half a million lines of JS), Google Docs, Google Maps, and Google Search. So it is no wonder that these teams encountered the difficulties of building a large JS application and strived for a better platform. Due to the fundamental flaws of JS and its slow evolution, something better was needed. A first attempt was **Google Web Toolkit (GWT)** where development was done in Java, which was then compiled to JS. Although reasonably successful because it enabled a structured and toolled approach to application building, again it was clear that the use of Java is somewhat awkward in a web environment. Thus arose the idea for Dart: a kind of hybrid platform between the dynamic nature of JS and the structured and toolable languages such as Java and C#. In order for Dart to run in all modern web browsers, as for GWT, it must be compiled to JS. Google has provided a special build of Chromium, called Dartium, that provides a **Dart virtual machine (VM)** to execute Dart code on-the-fly without any compilation step (this VM will soon be incorporated into Chrome; for the time being Chrome can be used to test the JS version of your Dart app).

Advantages of Dart

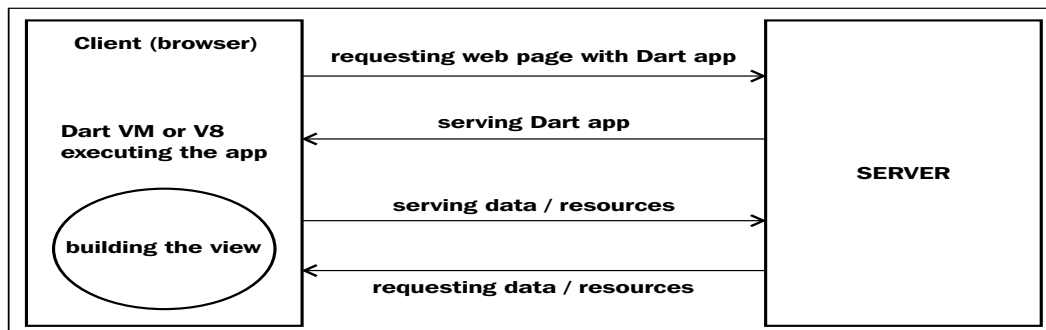
That way Dart can get a better performance profile than JS (remember that the same experts who developed the V8 JS VM are forging Dart, see <https://www.dartlang.org/performance/>), and at the same time maintain the simple and rapid development process of JS in the browser: the edit code, save, and refresh browser cycle to view the latest version, rather than having to stop, recompile, and run for every little change. Dart delivers high performance on all modern web browsers and environments ranging from small handheld devices to server-side execution. When it runs on its own VM, Dart is faster than JS (in Dart v1.0 around two times the performance of JS). Moreover, through **snapshotting** (a mechanism inherited from Smalltalk) a Dart app has a fast application startup time, in contrast to JS where all the source code has to be interpreted or compiled from source.

Dart can execute in the browser, either in its own Dart VM (only in Chrome for the moment) or compiled to JS, so Dart runs everywhere JS does. The Dart VM can also run standalone on a client or server.

Another big advantage compared with GWT is that Dart is much better integrated with the web page and like JS can directly manipulate the page elements and the document structure, that is, the **Document Object Model (DOM)**. Like JS, it has intimate access to the new HTML5 APIs, for example, drawing with the canvas, playing audio and video clips, or using the new local storage possibilities. Following the RIA model mentioned earlier, Dart executes the full application code in the browser, requesting data from the server and rebuilding the page user interface when needed. Because Dart wants to be part of the web, not just sit on top, the team has also built a Dart to JavaScript interop layer, to call JavaScript from Dart and the other way around. Together with its out-of-browser and server capabilities, Dart is also conceived for building complex, large-scale web applications. This can be clearly seen from its object-oriented nature, and Dart code is built with code clarity and structure (using libraries and packages) in mind.

To summarize:

- Dart compiles to JavaScript
- When run on its VM, Dart is faster than JavaScript
- Dart is better suited for large-scale applications



The Dart web model

Getting started with Dart


The Dart project team wants us to get an optimal development experience, so they provide a full but lightweight IDE: the **Dart Editor**, a light version of the well-known Eclipse environment. Installing this is the easiest way to get started, because it comprises the full Dart environment.

Installing the Dart Editor


Because Eclipse is written in Java, we need a Java Runtime Environment or JRE (Version 6 or greater) on our system (this is not needed for Dart itself, only for the Dart Editor). To check if this is already the case, go to <http://www.java.com/en/download/installed.jsp>.

If it is not the case, head for <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, and click on the **JRE DOWNLOAD** button, choose the JRE for your platform and then click on **Run** to start the installation.

Then go to <https://www.dartlang.org/> and click on the appropriate **Download Dart** button and under "Everything you need" choose the appropriate button (according to whether your installed OS is 32 bit or 64 bit) to download the editor.

 This is also the page where you can download the SDK separately, or the Dartium browser (a version of Chrome to test your Dart apps) or download the Dart source code.

You are prompted to save a file named `darteditor-os-xnn.zip`, where `os` can be `win32`, `linux`, or `macos`, and `nn` is 32 or 64. Extracting the content of this file will create a folder named `dart` containing everything you need: `dart-sdk`, `dartium`, and `DartEditor`. This procedure should go smooth but if you encounter a problem, please review <https://www.dartlang.org/tools/editor/troubleshoot.html>.

 In case you get the following error message: **Failed to load the JNI shared library C:\Program Files(x86)\Java\jre6\bin\client\jvm.dll**, do not worry. This happens when JRE and Dart Editor do not have the same bit width. More precisely, this happens when you go to www.java.com to download JRE. In order to be sure what JRE to select, it is safer to go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, click on the **JRE DOWNLOAD** button and choose the appropriate version. If possible, use a 64-bit versions of JRE and Dart Editor.

Your first Dart program

Double-click on `DartEditor.exe` to open the editor. Navigate to **File | New Application** or click on the first button below the menu (**Create a new Dart Application...**). Fill in an application name (for example, `dart1`) and choose the folder where you want the code file to be created (make a folder such as `dart_apps` to provide some structure; you can do this while using the **Browse** button). Select **Command-line application**.

With these names a folder `dart1` is made as a subfolder of `dart_apps`, and a source-file `dart1.dart` is created in `dart1\bin` with the following code (we'll explain the `pubspec.yaml` and the `packages` folder in one of the following examples):

```
void main() {  
    print("Hello, World!");  
}
```



Downloading the example code

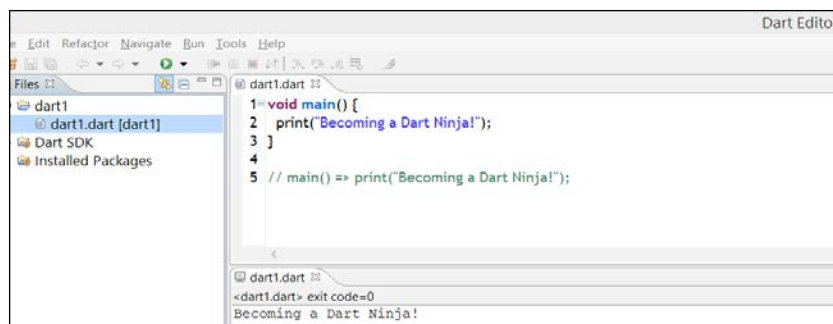
You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Here we see immediately that Dart is a C-syntax style language, using `{ }` to surround code and `;` to terminate statements. Every app also has a unique `main()` function, which is the starting point of the application.

This is probably the shortest app possible, but it can be even shorter! The keyword `void` indicates (as in Java or C#) that the method does not explicitly return an object (indeed a `print` only produces output to the console), but return types can be left out. Furthermore, when a function has only one single expression, we can shorten this further to the following elegant shorthand syntax:

```
main() => print("Hello, World!");
```

Now, change the printed string to `"Becoming a Dart Ninja!"` and click on the green arrow button (or press `Ctrl + R`) to run the application. You should see something like the following screenshot (where the **Files**, **Apps**, and **Outline** items from the **Tools** menu were selected):



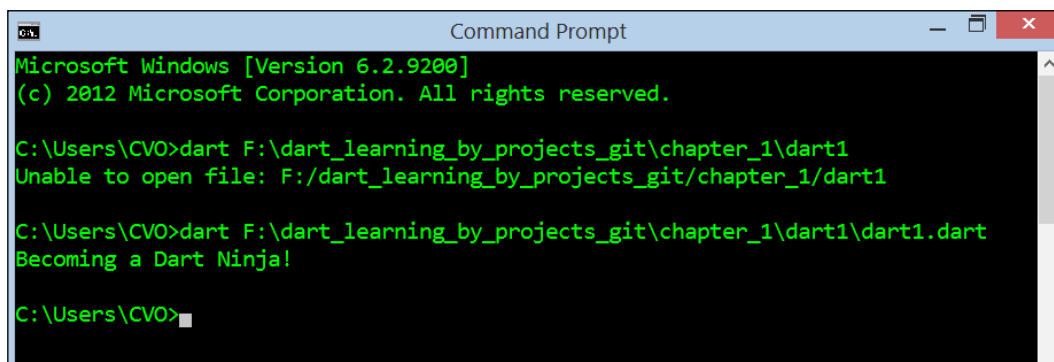
The Dart Editor

You have successfully executed your first Dart program!

Near the bottom of the screen we see our string printed out, together with the message `exit code=0` meaning all went well.

The **Files** tab is useful for browsing through your applications, and for creating, copying, moving, renaming, and deleting files. The **Outline** tab (available via **Tools | Outline**) now only shows `main()`, but this tool will quickly become very useful because it provides an overview of the code in the active file.

Because this was a command-line application, we could just as easily have opened a console in our folder `dart1` and executed the command: `dart dart1.dart` to produce the same output as shown in the following screenshot:



A Dart console application



To let this work, you must first let the OS know where to find the dart VM; so, for example, in Windows you change the `PATH` environment variable to include `C:\dart\dart-sdk\bin`, if your Dart installation lives in `C:\dart`.

Getting a view on the Dart tool chain

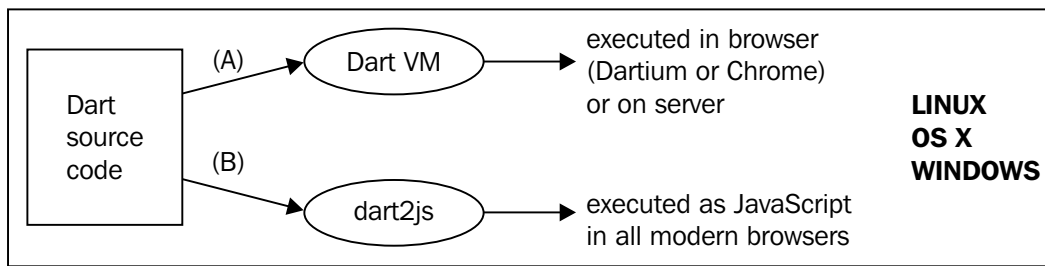
Dart comes with *batteries included*, which means that a complete stack of tools is provided by Google to write Dart apps, compile, test, document, and publish them. Moreover, these tools are platform independent (being made for 32- and 64-bit Linux, OS X, and Windows) and they are integrated in the Dart Editor IDE. The Dart Editor contains everything a seasoned developer needs to work with confidence on his app:

- Syntax coloring for keywords
- Autocompletion for class members (by typing `.` after a name you get a list of available properties and methods)
- Folding/unfolding code blocks
- Tools for navigating the code (a handy overview of the code with the Outline, find callers of a method, and so on)
- Full debugging capabilities of both browser and server applications
- Choose your preferred editor style by navigating to **Tools | Preferences | Visual Theme**
- Quick fixes for common errors
- Refactoring capabilities
- Direct access to the online API documentation by navigating to **Help | API Reference**

The code you make is analyzed while you type, indicating warning (yellow triangles) or errors (red underscores or stop signs). To get more acquainted and experiment with these possibilities, go and read the documentation at <https://www.dartlang.org/tools/editor/> and play with one of the samples such as Sunflower or Pop, Pop, Win! (you can find the samples by navigating to **Tools | Welcome Page**). From now on use the editor in conjunction with the code examples of the book, so that you can try them out and test changes.

The Dart execution model

How a Dart app executes is sketched in the following diagram:



Dart execution model

The Dart code produced in the Dart Editor (or in a plugin for Eclipse or IntelliJ) can:

- Execute in the **Dart VM**, hosted in Dartium or Chrome (Dartium is an experimental version of Chrome to test out Dart) or directly in the operating system (the browser VM knows about HTML, the server VM does not, but can use, for example, IO and sockets, so they are not completely equivalent)
- Be compiled to JS with the dart2js compiler, so that it can run in all recent browsers

Code libraries in Dart are called packages and the core **Dart SDK** contains the basic types and functionalities for working with `collection`, `math`, `html`, `uri`, `json`, and so on. They can be recognized by the syntax `dart:prefix`, for example, `dart:html`. If you want to use a functionality from a library in a code file, you must import it by using the following as the first statement(s) in your code (`dart:core` is imported by default):

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

The Dart code can be tested with the **unit test** package, and for documentation you can use the **dartdoc** tool (which you can find by navigating to **Tools | Generate Dartdoc** in Dart Editor), which generates a local website structured like the official API documentation on the Web. The **pub** tool is the Dart package manager: if your app needs other packages besides the SDK, pub can install them for you (from the **Tools** menu item in Dart Editor, select **Pub Get** or **Pub Upgrade**) and you can also publish your apps with it in the web repository <http://pub.dartlang.org/>.

We will see all of these tools in action in *Chapter 2, Getting to Work with Dart*.

A bird's eye view on Dart

It's time to get our feet wet by working on a couple of examples. All code will be thoroughly explained step by step; along the way we will give you a lot of tips and in the next chapter we will go into more detail on the different possibilities, thus gaining deeper insight into Dart's design principles.

Example 1 – raising rabbits

Our first real program will calculate the procreation rate of rabbits, which is not only phenomenal but indeed exponential. A female rabbit can have seven litters a year with an average of four baby rabbits each time. So starting with two rabbits, at the end of the year you have $2 + 28 = 30$ rabbits. If none of the rabbits die and all are fertile, the growth rate follows the following formula, where n is the number of rabbits after the years specified:

$$n(\text{years}) = 2 \times e^{(k \times \text{years})}$$

Here the growth factor $k = \ln(30/2) = \ln 15$. Let us calculate the number after each year for the first 10 years.

Go to **File | New Application** as before, select **Command-line application** and type the following code, or simply open the script from `chapter_1` in the provided code listings. (Don't worry about the file `pubspec.yaml`; we'll discuss it in the web version.)

The calculation is done in the following Dart script `prorabbits_v1.dart`:

```
import 'dart:math' (1)

void main() {
  var n = 0; // number of rabbits (2)

  print("The number of rabbits increases as:\n"); (3)
  for (int years = 0; years <= 10; years++) { (4)
    n = (2 * pow(E, log(15) * years)).round().toInt(); (5)
    print("After $years years:\t $n animals"); (6)
  }
}
```

Our program produces the following output:

The number of rabbits increases as:

```
After 0 years:    2 animals
After 1 years:   30 animals
After 2 years:  450 animals
After 3 years:  6750 animals
After 4 years: 101250 animals
After 5 years: 1518750 animals
After 6 years: 22781250 animals
After 7 years: 341718750 animals
After 8 years: 5125781250 animals
After 9 years: 76886718750 animals
After 10 years: 1153300781250 animals
```

So if developing programs doesn't make you rich, breeding rabbits will. Because we need some mathematical formulas such as natural logarithms `log` and power `pow`, we imported `dart:math` in line (1). Our number of livestock `n` is declared in line (2); you can see that we precede its name with `var`. Here, we don't have to indicate the type of `n` as `int` or `num` (so called *type annotations*), as Dart uses *optional typing*.



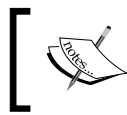
Local variables are commonly declared untyped as `var`.

We could have declared it to be of type `num` (number) or `int`, because we know that `n` is a whole number. But this is not necessary as Dart will derive that from the context in which `n` is used. The other `num` type is called `double`, used for decimal numbers. Also the initialization part (`= 0`) could have been left out. With no initialization `var n`; or even `int n`; gives `n` the value `null`, because every variable in Dart is an object. The keyword `null` simply indicates that the object has no value yet (meaning it is not yet allocated in heap memory). It will come as no surprise that `//` indicates the beginning of a comment, and `/*` and `*/` can be used to make a multi-line comment.



Comment a section of code by selecting it and then click on **Toggle comment** in the **Edit** menu.

In lines (3) and (6) we see that within a quoted string we can use escape characters such as `\n` and `\t` to format our output. Line (4) uses the well-known for-loop that is also present in Dart. In order to have the count of animals as a whole number we needed to apply the `round` function. The `pow` function produces a `double` and because 6750.0 animals doesn't look so good, we have to convert the `double` to an `int` with the `toInt()` function. In line (6), the elegant string substitution mechanism (also called string interpolation) is used: `print` takes a string as argument (a string variable: any expression enclosed within `" "` or `' '`) and in any such quoted string expression you can substitute the value of variable `n` by writing `$n`. If you want the value of an expression within a string, such as `a + b`, you have to enclose the expression with braces, for example, `${a + b}`.



You don't have to write the `${n}` when displaying a variable `n`; just use `$n`. You can also simply use `print(n)`.

It is important to realize that we did not have to make any class in our program. Dart is no class junkie like Java or C#. A lot can be done only with functions; but if you want to represent real objects in your programs, classes is the way to go (see the *Example 2 – banking* section).

Extracting a function

This version of our program is not yet very modular; we would like to extract the calculation in a separate method `calculateRabbits(years)` that takes the number of years as a parameter. This is shown in the following code (version 2 line (4) of `prorabbits_v2.dart`) with exactly the same output as version 1:

```
import 'dart:math';

int rabbitCount = 0;
const int NO_YEARS = 10;
const int GROWTH_FACTOR = 15;

void main() {
  print("The number of rabbits increases as:\n");
  for (int years = 0; years <= NO_YEARS; years++) {
    rabbitCount = calculateRabbits(years);
    print("After $years years:\t $rabbitCount animals");
  }
}

int calculateRabbits(int years) {
  return (2 * pow(E, log(GROWTH_FACTOR) *
    years)).round().toInt();
}
```

We could have written this new function ourselves, but Dart has a built-in refactoring called **Extract Method**. Highlight the line:

```
n = (2 * pow(E, log(15) * years)).round().toInt();
```



Right-click and select **Extract Method**. Dart will do the bulk of the work for you, but we can still simplify the proposed code by omitting the parameter `n`.

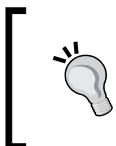
The `calculateRabbits` function calculates and returns an integer value; this is indicated by the word `int` preceding the function name. We give the function a type here because it is top level, but the program would have run without the function-type indication.

This new function is called by `main()`. This is the way a Dart program works: all lines in `main()` are executed in sequence, calling functions as needed, and the execution (and with it the Dart VM) stops when the ending `}` of `main()` is reached. We rename the variable `n` to `rabbitCount`, so we need no more comments.



Renaming a variable is also a built-in refactoring. Select the variable (all occurrences are then indicated), right-click, and select **Rename**.

A good programmer doesn't like hardcoded values such as 10 and 15 in a program; what if they have to be changed? We replace them with constant variables, indicated with keyword `const` in Dart, whose name is, by convention, typed in capital letters and parts separated by `_`, see lines (2) and (3).



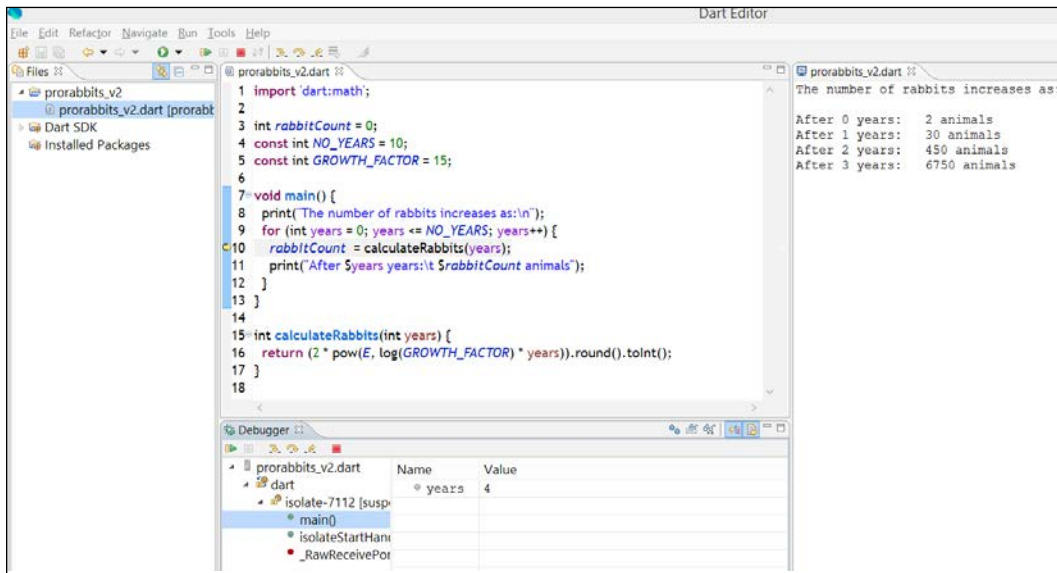
Take care of your top-level variables, constants, and functions because they will probably be visible outside your program (sometimes called the interface or API of your program); type them and name them well.

And now for some practice:

1. Examine this second version by going to **Tools | Outline**.
2. Set a breakpoint on the line `rabbitCount = calculateRabbits(years);` by double-clicking in the margin in front.
3. Run the program and learn how to use the features of the Debugger tool (Press `F5` to step line by line, `F6` or `F7` to step over or out of a function, and `F8` to resume execution until the next breakpoint is hit).

4. Watch the values of the `years` and `rabbitCount` variables.

The output should resemble the following screenshot:



Debugging `prorabbits_v2.dart`

A web version

As a final version for now, let us build an app that uses an HTML screen where we can input the number of years of rabbit elevation and output the resulting number of animals. Go to **File | New Application**, but this time select **Web application**. Now a lot more code is generated that needs explaining. The app now contains a subfolder `web`; this will be the home for all of the app's resources, but for now it contains a stylesheet (`.css` file), a hosting web page (`.html`), and a startup code file (in our case `prorabbits_v3.dart`). The first line in this file makes HTML functionality available to our code:

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

We remove the rest of the example code so only an empty `main()` function remains. Look at the source of the HTML page, right before the `</body>` tag; it contains the following code:

```
<script type="application/dart" src="prorabbits_v3.dart"></script>
<script src="packages/browser/dart.js"></script>
```


The first line is evident: our Dart script must be started. But wait, how do we know that there is a Dart VM available in this browser? This will be checked in the second JavaScript file, `dart.js`; the first few lines of code in this file are:

```
if (navigator.webkitStartDart) {
  // Dart VM is available, start it!
} else {
  // Fall back to compiled JavaScript
}
```

The Dart VM exists for the moment only in Dartium (soon in Chrome). For other browsers we must supply the Dart-to-JS compiled scripts; this compilation can be done in the Editor by navigating to **Tools | Generate Javascript**. The output size is minimal: *dead* js code that is not used is eliminated in a process called tree shaking. But where does this mysterious script `dart.js` come from? `src="packages/browser/dart.js"` means that it is a package available in the Dart repository <http://pub.dartlang.org/>.

External packages that your app depends on need to be specified in the section, *dependencies*, in the file `pubspec.yaml`. In our app this section contains the following parameters:

```
name: prorabbits_v3
description: Raising rabbits the web way
dependencies:
  browser: any
```

We see that our app depends on the browser package; any version of it is OK. The package is added to your app when you right-click on the selected `pubspec.yaml` and select **Pub Get**: a folder `packages` is added to your app, and per package a subfolder is added containing the downloaded code, in our case `dart.js`. (In *Chapter 2, Getting to Work with Dart*, we will explore pub in greater depth.)

For this program we replace the HTML `<p id="sample_text_id"></p>` as shown in the following code:

```
<input type="number" id="years" value="5" min="1" max="30">
<input type="button" id="submit" value="Calculate"/>
<br/>Number of rabbits: <label id="output"></label>
```

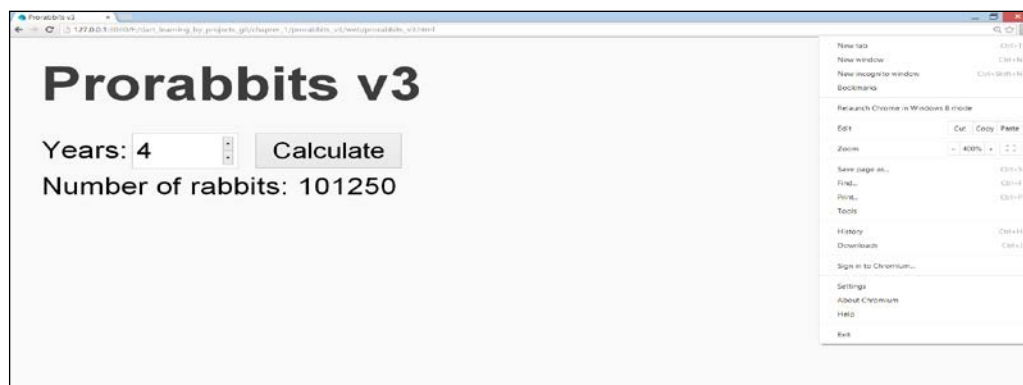
The input field with type number (new in HTML5) gives us a NumericUpDown control with a default value 5 and limited to the range 1 to 30. In our Dart code, we now have to handle the click-event on the button with id as submit. We do this in our main() function with the following line of code:

```
query Selector("#submit").onClick.listen( (e) => calcRabbits() );
```

query Selector("#submit") gives us a reference in the code to the button, listen redirects to an anonymous function (see *Chapter 2, Getting to Work with Dart*) to handle this event e, which calls the function calcRabbits() shown in the following code:

```
calcRabbits() {  
  // binding variables to html elements:  
  InputElement yearsInput = querySelector("#years");           (1)  
  LabelElement output = querySelector("#output");               (2)  
  // getting input  
  String yearsString = yearsInput.value;  
  int years = int.parse(yearsString);  
  // calculating and setting output:  
  output.innerHTML = "${calculateRabbits(years)}";  
}
```

Here in lines (1) and (2), the input field and the output label are bound to the variables in_years and output. This is always done in the same way: the query Selector() function takes as its argument a CSS-selector, in this case the ID of the input field (an ID is preceded by a # sign). We typed in_years as an InputElement (because it is bound to an input field), that way we can access its value, which is always a string. We then convert this string to an int type with the function int.parse(), because calculateRabbits needs an int parameter. The result is shown as HTML in the output label via string substitution, see the following screenshot:



The screen of prorabbits_v3

All objects in Dart code that are bound to HTML elements are instances of the class `Element`. Notice how you can change the Dart and HTML code; save and hit refresh in Dartium (Chrome) to get the latest version of your app.

Example 2 – banking

All variables (strings, numbers, and also functions) in Dart are objects, so they are also instances of a class. The class concept is very important in modeling entities in real-world applications, making our code modular and reusable. We will now demonstrate how to make and use a simple class in Dart modeling a bank account. The most obvious properties of such an object are the owner of the account, the bank account number, and the balance (the amount of money it contains). We want to be able to deposit an amount of money in it that increases the balance, or withdrawing an amount so as to decrease the balance. This can be coded in a familiar and compact way in Dart as shown in the following code:

```
class BankAccount {
  String owner, number;
  double balance;
  // constructor:
  BankAccount(this.owner, this.number, this.balance);    (1)
  // methods:
  deposit(double amount) => balance += amount;          (2)
  withdraw(double amount) => balance -= amount;
}
```

Notice the elegant constructor syntax in line (1) where the incoming parameter values are automatically assigned to the object fields via `this`. The methods (line (2)) can also use the shorthand `=>` function syntax because the body contains only one expression. If you prefer the `{ }` syntax, they will be written as follows:

```
deposit(double amount) {
  balance += amount;
}
```

The code in `main()` makes a `BankAccount` object `ba` and exercises its methods (see program `banking_v1.dart`):

```
main() {
  var ba = new BankAccount("John Gates",
    "075-0623456-72", 1000.0);
  print("Initial balance:\t\t ${ba.balance} \t");
  ba.deposit(250.0);
  print("Balance after deposit:\t\t ${ba.balance} \t");
  ba.withdraw(100.0);
  print("Balance after withdrawal:\t\t ${ba.balance} \t");
}
```

The preceding code produces the following output:

```
Initial balance:           1000.0 $
Balance after deposit:     1250.0 $
Balance after withdrawal:  1150.0 $
```

Notice how when you type `ba.` in the editor, the list of `BankAccount` class members appears to autocomplete your code. By convention, variables (objects) and functions (or methods) start with a lower case letter and follow the camelCase notation (<http://en.wikipedia.org/wiki/CamelCase>), while class names start with a capital letter, as well as the word-parts in the name. Remember Dart is case sensitive!

Making a todo list with Dart

Since this has become the "Hello World" for web programmers, let's make a simple todo list and start a new web application `todo_v1`. To record our tasks we need an input field corresponding with `InputElement` in Dart:

```
<input id="task" type="text" placeholder=
  "What do you want to do?"/>
```

The HTML5 placeholder attribute lets you specify default text that appears in the field.

We specify a list tag (`UListElement`) that we will fill up in our code:

```
<ul id="list"/>
```

The following is the code from `todo_v1.dart`:

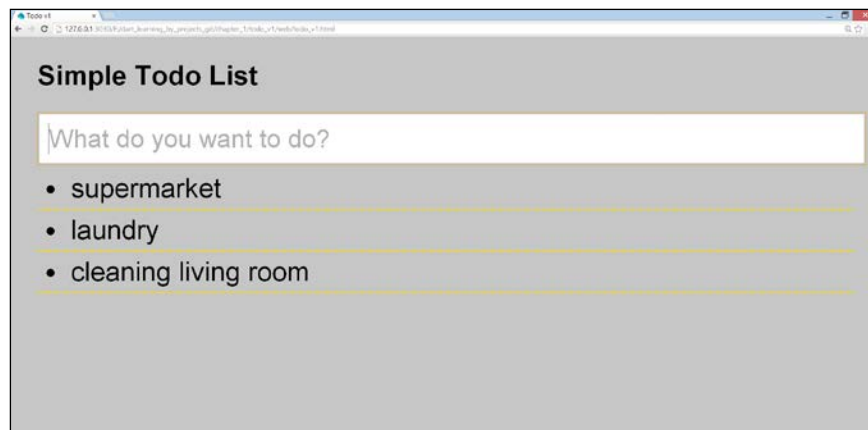
```
import 'dart:html';

InputElement task;
UListElement list;

main() {
  task = querySelector('#task');           (1)
  list = querySelector('#list');           (2)
  task.onChange.listen( (e) => addItem() ); (3)
}

void addItem() {
  var newTask = new LIElement();          (4)
  newTask.text = task.value;               (5)
  task.value = '';                         (6)
  list.children.add(newTask);              (7)
}
```

We bind our HTML elements to the Dart objects `task` and `list` in lines (1) and (2). In line (3) we attach an event-handler `addItem` to the `onChange` event of the textfield `task`: this fires when the user enters something in the field and then leaves it (either by pressing *Tab* or *Enter*). `UListElement` is in fact a collection of `LIElements` (these are its children); so for each new task we make a `LIElement` (4), assign the task's value to it (5), clear the input field (6), and add the new `LIElement` to the list in (7). In the following screenshot you can see some tasks to be performed:



A simple todo list

Of course this version isn't very useful (unless you want to make a print of your screen); our tasks aren't recorded and we can't indicate which tasks are finished. Don't worry; we will enhance this app in the future versions.

Summary

We covered a lot of ground in this introductory chapter, but by now you know the case for Dart in the context of web applications, where Dart apps can live and how they are executed, and the various tools to work with Dart, in particular the Dart Editor.

You also got acquainted with some simple command line and web Dart apps and got a feeling for the Dart syntax. In the next chapter, we explore the various code and data structures of Dart more systematically and any obscurities that are still there in your mind will surely disappear. More coming soon to a Dart center near you...

