The Dart Programming Language

Gilad Bracha

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Dart Programming Language

This page intentionally left blank

The Dart Programming Language

Gilad Bracha

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2015953614

ISBN-13: 978-0-321-92770-5
ISBN-10: 0-321-92770-2
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

To my mother, Shoshana,
who taught me to be picky.

This page intentionally left blank

Contents

Foreword

In the early spring of 2006, I wrote a short blog post called "Gilad is Right" where, as a recovering typaholic, I admitted that Gilad's idea of optional and layered type systems, where static types cannot change the runtime behavior of the program and do not prevent an otherwise legal program from compiling or executing, was a necessary design trade-off for programming languages aimed at millions of developers. At that time I was working on Visual Basic, which already supported a form of optional typing by means of the Option Strict Off statement, but that feature was under heavy fire from static typing proponents. Type systems are often highly non-linear and after a certain point their complexity explodes while adding very little value to the developer and making life miserable for the language implementors. Optional and layered type systems enable a much more gradual approach by allowing strong static typing to coexist peacefully with dynamic typing. Now nearly a decade later, the vision Gilad pioneered has become mainstream under the name *gradual typing*. Many programming languages that have been created in the last few years, such as Hack, TypeScript, Flow, Racket, and of course Dart, are gradually typed. Even academics have embraced the idea and write papers about it with frivolous titles that include words such as "threesomes" and "blame."

Another pragmatic aspect of Dart, but one that language purists have not yet accepted, is the fact that the Dart type system is deliberately unsound. In normal English this means that the Dart type checker will not flag certain type errors at compile time, but relies on runtime checking instead to ensure type safety. The main source of type unsoundness in Dart is covariant generics. To explain what variance is, let's first look at a vending machine from which we can only take drinks. If a cafeteria requires a vending machine with soda pop, we can legally install a vending machine that dispenses root beer since root beer is a type of soda pop (but it is illegal to install a vending machine for soda pop where a vending machine for root beer is required). In programming language speak we say that vending machines are *covariant*. Next let's take a look at garbage cans into which we can throw only garbage. If a cafeteria requires a garbage can for recyclables, we can legally install a garbage can for trash since recyclable garbage is a type of trash (but it is illegal to install a garbage can for recyclables where a garbage can for trash is required). In programming language speak we say that garbage cans are *contravariant*. If you are a little puzzled about contravariance you are not the only one, and you will appreciate Dart's decision to make all generic types covariant. The consequence of that choice is that if you need a garbage can for trash, you can legally install a garbage can for recyclables, but that garbage can will reject all non-recyclable

trash that people are trying to dump in it. While theoretically unsound, unsafe variance actually feels rather natural for most developers, and I applaud the choice the Dart designers made here. As anyone that has struggled with ? **super** and ? **extends** can attest, languages that have chosen in favor of static type safety for generics do so at the expense of their users.

The Dart language designers made additional pragmatic choices that make coding in Dart a smooth experience. For example Dart has no interfaces, abstract base classes, or "normal" classes. Instead Dart only has classes that can be used as interfaces by implementing them, or used as base classes by extending them, or have their implementation reused by mixing them in. Every type in Dart is an object, so there is no difference between primitive (e.g., numeric) types and regular object types. Even though everything in Dart is an object, it is possible to define top-level functions and variables, so one no longer needs the dreaded **public static void** main incantation inside a top-level class to get going. Dart allows user-defined arithmetic operators, but does not support type-based overloading of methods. This significantly simplifies the language. In other programming languages that do support type-based overloading, the exact semantics of that feature often take up an unjustifiably large fraction of the language specification. Null-aware operators (even **null** is a normal object) and cascades give more power to the dot and make every API fluent for the user with little effort from the API writer.

While Dart is essentially a dynamic language because all types are optional, you will encounter far fewer "wat" moments than with most other dynamic languages. There is **null** but no undefined, and hence only == but no ===. Only **true** is true, so no more (foo && foo.bar()) to check for **null**. Dart has regular integer and floating point numeric types, without the mind-blowing runtime type conversions around + and == that make great exam questions, entertaining conference presentations, but frustrating bugs.

In my opinion, though obviously I am biased, what puts Dart right at the top of my favorite programming languages is that it is the only language that I know of that supports all four essential effects of programming:

	One	Many
sync	{... **return** *e*; ... }	**sync*** {... **yield** *e*; ... **for**() ... }
async	**async** {... **await** *e* ... }	**async*** {... **await** *e* ... **yield** *e*; ... } **async*** {... **await for**() ... }

That is, Dart has deep support for producing and consuming synchronous data streams (Iterable<T>) using generators and for loops inside **sync*** blocks, producing and consuming futures (Future<T>) using **await** expressions inside **async** blocks, and last but not least support for producing and consuming asynchronous data streams (Stream<T>) using asynchronous generators and for loops inside **async*** blocks. Built-in support for asynchronous programming is essential in any modern programming language, where even data in memory, let alone data across the network, is "far away" and imposes such high latency that synchronous access is prohibitively expensive. Like JavaScript, but unlike other languages that support generators, Dart has so-called delegating generators that avoid quadratic blowup of nested and recursively generated streams.

Despite all these nice touches, Dart is essentially a rather boring language by design. Thanks to support for getters, setters, lambdas, enums, reified generics, modules, an extensive well-crafted standard library, and a snappy package manager, Dart will feel comfortable, like well-worn shoes, if you are a developer coming from Java or C#, and feel like a breath of fresh air when coming from Javascript. This book will help you to understand the why, how, and what of all of Dart's features in Gilad's signature painstaking detail and inimitable style and get you productive in Dart in no time.

Erik Meijer
Palo Alto, California
October 2015

This page intentionally left blank

Preface

How is this book different from other books on Dart? Other books on Dart are full of practicalities; this book is focused on principles and ideas.

The practicalities of Dart are very important, but they are different this year than they were last year, and will likely differ again the year afterwards. In contrast, the principles behind Dart should be more applicable over time. You should read this book if you are interested in the ideas that motivate the design of the language, and how they manifest in the pragmatic, demanding real-world setting that is Dart.

One of the chief ideas in Dart is optional typing. I started working on optional types decades ago; today we see a proliferation of optional type systems, which I find immensely satisfying. While neither Dart nor any of its competitors realize optional types exactly as I would like them to, the fact that the idea has hit the mainstream is what matters.

Even more important is the idea that Dart is an object-oriented language, not in the conventional sense of classes, inheritance and the other standard paraphernalia known to most programmers, but in the deep sense that only the observable behavior of an object matters. Again, this idea is realized imperfectly, but better than in most mainstream languages.

Another key idea covered in this book is reflection. The topic of reflection is not well addressed by books on programming. For that reason, I was very keen to discuss reflection in this book. However, the history of reflection in Dart has been unexpectedly tortuous.

On the one hand, many of Dart's users have been eager to use reflection, sometimes for purposes it is not ideal for. On the other hand, certain current platforms are severely limited as compilation targets for Dart, and make reflection support relatively expensive, especially with respect to program size. This tension put Dart reflection in an awkward spot.

The sensitivity to code size and reflection's effect on it were well understood from the beginning. The problem and its solutions were discussed in the very first design document for Dart reflection in November 2011. Nevertheless it took some four years until the solution was realized in a form that programmers could easily apply.

I hope this book conveys these ideas and others effectively, but that is for you, the reader, to judge. It is possible that one could do better using a more purist language, but on the other hand it's not clear one would reach as large an audience. Perhaps someday I will try that experiment and see.

This book has been a long time coming. I delayed finishing the book until I could tell a reasonably cohesive story about reflection. Another reason for the delay is that the book's topic has evolved so rapidly that it was constantly at risk of being out of date. That risk has not passed, but at some point one needs to say "enough is enough."

Dart is an imperfect realization of the ideas that drove its design. No one is more aware of this fact than its designers. Nevertheless, it is a real language, in which millions of lines of mission-critical code have been written. It has moved the culture of programming forward in some ways, most notably in the area of optional typing. As they say in Denmark: It could have been worse.

Acknowledgments

The Dart programming language is the result of a large team effort. My work on Dart has been made much more pleasant because it has involved many colleagues who have been a pleasure to work with, many of whom are not only co-workers but friends. It has been my privilege to document some of this effort in this book, as well as via the Dart Language Specification.

The Dart language was conceived and designed by Lars Bak and Kasper Lund, and so this book literally could not exist without them. Lars is a longtime friend and colleague who has led the Dart project from its beginning. Among other things, Lars got me involved with Dart, and for that I owe him special thanks. Both Lars and Kasper are not only phenomenally talented systems designers and implementors, but fun to be with as well!

Special thanks also to Erik Meijer; working with him on the asynchronous features of Dart was a joy. Erik is a real programming language professional, of a caliber one meets only rarely.

If Chapter 8 owes a lot to Erik, Chapter 7 owes much to Ryan Macnak, who implemented mirrors in the Dart VM, to Peter Ahé, who pioneered them in dart2js, and to Erik Ernst, who worked on the reflectable library.

My work on the book you hold in your hands was supported not only by Lars, but also by my manager Ivan Posva. For the past four years I have shared office space with Ivan and the other members of the VM team in Mountain View: Zachary Anderson, Daniel Andersson, Siva Annamalai, Régis Crelier, Matthias Hausner, Ryan Macnak, John McCutchan, Srdjan Mitrovic and Todd Turnidge. I thank them for the pleasant company.

My frequent visits to Dart supreme headquarters in Aarhus, Denmark, have always been fun (even if the actual travel was not). The administrative support of Linda Lykke Rasmussen has been priceless.

My work on the Dart specification is the direct basis for this book. That work has benefited from the careful critiques of many people, but none more than Lasse Nielsen whose phenomenal attention to detail has caught many subtle issues.

I've also been heavily involved in the process of standardizing Dart. Anders Sandholm has shielded me from much of the burdens involved therein; I owe him for that. I also thank the other participants in the Dart standards committee, ECMA TC52.

Dart would not be possible without the work of many other Dart team members past and present. They are too numerous to list but they have all contributed to making Dart what it is today.

My longtime editor, Greg Doench, has always been exceedingly patient and a pleasure to work with.

As always, my wife, Weihong, and my son, Teva, make it all worthwhile.

Gilad Bracha
Los Altos, California
November 2015

About the Author

Gilad Bracha is a software engineer at Google where he works on Dart. In prior lives he has been a VP at SAP Labs, a Distinguished Engineer at Cadence, and a Computational Theologist and Distinguished Engineer at Sun. He is the creator of the Newspeak programming language, co-author of the Java Language and Virtual Machine Specifications, and a researcher in the area of object-oriented programming languages. Prior to joining Sun, he worked on Strongtalk, the Animorphic Smalltalk System. He received his B.Sc. in Mathematics and Computer Science from Ben Gurion University in Israel and a Ph.D. in Computer Science from the University of Utah.

This page intentionally left blank

Chapter 1

Introduction

Dart is a general purpose programming language. It is a new language in the C tradition, designed to be familiar to the vast majority of programmers. The obligatory "Hello World" example illustrates how familiar Dart syntax is:

```
main(){
  print('Hello World');
}
```

Unless your background in programming and computer science is either extremely unusual, or lacking entirely, this code should be virtually self-explanatory. We will of course elaborate on this program and more interesting ones in the pages that follow.

Dart is purely object-oriented, class-based, optionally typed and supports mixin-based inheritance and actor-style concurrency. If these terms are unfamiliar, fear not, they will all be explained as we go along.

That said, this book is not intended as a tutorial for novices. The reader is expected to have a basic competence in computer programming.

While the bulk of this book will describe and illustrate the semantics of Dart, it also discusses the rationale for certain features. These discussions are included because, in my experience, good programmers are interested not only in what a programming language does, but why. And so, the next few sections give a very high level overview of the philosophy behind Dart. Later sections will also incorporate discussions of design decisions, alternatives and the history of the key ideas. However, if you are eager to just dive in, section 1.4 gives a quick tutorial.

And now, on with the show!

1.1 Motivation

The rise of the world-wide web has changed the landscape of software development. Web browsers are increasingly seen as a platform for a wide variety of software applications. In recent years, mobile devices such as smartphones and tablets have also become increasingly ubiquitous. Both of these trends have had a large impact on the way software is written.

Web browsers started as a tool to display static hypertext documents. Over time, they evolved to support dynamic content. Dynamic content is computed and re-computed over time and has grown from simple animations to server-based applications such as database front-ends and store fronts for internet commerce to full-fledged applications that can run offline.

This evolution has been organic; a series of accidents, some happy and some less so, have enabled such applications to run on an infrastructure that was not really designed for this purpose.

Mobile applications pose their own challenges. These applications must conserve battery life, providing a new incentive to improve performance. Network access may be slow, costly or even absent. Mobile platforms tend to impose a particular life cycle with particular restrictions on size.

Dart is intended to provide a platform that is specifically crafted to support the kinds of applications people want to write today. As such it strives to protect the programmer from the undesirable quirks and low-level details of the underlying platform while providing easy access to the powerful facilities new platforms have to offer.

1.2 Design Principles

1.2.1 Everything Is an Object

Dart is a pure object-oriented language. That means that all values a Dart program manipulates at run time are objects—even elementary data such as numbers and Booleans. There are no exceptions.

Insisting on uniform treatment of all data simplifies matters for all those involved with the language: designers, implementors and most importantly, users.

For example, collection classes can be used for all kinds of data, and no one has to be concerned with questions of autoboxing and unboxing. Such low-level details have nothing to do with the problems programmers are trying to solve in their applications; a key role of a programming language is to relieve developers of such cognitive load.

Perhaps surprisingly, adopting a uniform object model also eases the task of the system implementor.

1.2.2 Program to an Interface, not an Implementation

The idea that what matters about an object is how it behaves rather than how it is implemented is the central tenet of object-oriented programming. Unfortunately this tenet is often ignored or misunderstood.

Dart works hard to preserve this principle in several ways, though it does so imperfectly.

- Dart types are based on interfaces, not on classes. As a rule, any class can be used as an interface to be implemented by other classes, irrespective of whether the two share implementation (there are a few exceptions for core types like numbers, Booleans and strings).

- There are no final methods in Dart. Dart allows overriding of almost all methods (again, a very small number of built-in operators are exceptions).
- Dart abstracts from object representation by ensuring that all access to state is mediated by accessor methods.
- Dart's constructors allow for caching or for producing instances of subtypes, so using a constructor does not tie one to a specific implementation.

As we discuss each of these constructs we will expand on their implications.

1.2.3 Types in the Service of the Programmer

There is perhaps no topic in the field of programming languages that generates more intense debate and more fruitless controversy than static typechecking. Whether to use types in a programming language is an important design decision, and like most design decisions, involves trade-offs.

On the positive side, static type information provides valuable documentation to humans and computers. This information, used judiciously, makes code more readable, especially at the boundaries of libraries, and makes it easier for automated tools to support the developer.

Types simplify various analysis tasks, and in particular can help compilers improve program performance.

Adherents of static typing also argue that it helps detect programming errors.

Nothing comes for free, and adding mandatory static type checking to a programming language is no exception. There are, invariably, interesting programs that are prohibited by a given type discipline. Furthermore, the programmer's workflow is often severely constrained by the insistence that all intermediate development states conform to a rigid type system. Ironically, the more expressive the type discipline, the more difficult it is to use and understand. Often, satisfying a type checker is a burden for programmers. Advanced type systems are typically difficult to learn and work with.

Dart provides a productive balance between the advantages and disadvantages of types. Dart is an optionally typed language, defined to mean:

- Types are syntactically optional.
- Types have no effect on runtime semantics.

Making types optional accommodates those programmers who do not wish to deal with a type system at all. A programmer who so chooses can treat Dart as an ordinary dynamically typed language. However, all programmers benefit from the extra documentation provided by any type annotations in the code. The annotations also allow tools to do a better job supporting programmers.

Dart gives warnings, not errors, about possible type inconsistencies and oversights. The extent and nature of these warnings is calibrated to be useful without overwhelming the programmer.

At the same time, a Dart compiler will never reject a program due to missing or inconsistent type information. Consequently, using types never constrains the developer's

workflow. Code that refers to declarations that are absent or incomplete may still be productively run for purposes of testing and experimentation.

The balance between static correctness and flexibility allows the types to serve the programmer without getting in the way.

The details of types in Dart are deferred until Chapter 5, where we explain the language's type rules and explore the trade-offs alluded to above in detail.

1.3 Constraints

Dart is a practical solution to a concrete problem. As such, Dart's design entails compromises. Dart has to run efficiently on top of web browsers as they exist today.

Dart also has to be immediately recognizable to working programmers. This has dictated the choice of a syntax in the style of the C family of programming languages. It has also dictated semantic choices that are not greatly at variance with the expectations of mainstream programmers. The goal has not been radical innovation, but rather gradual, conservative progress.

As we discuss features whose semantics have been influenced by the above constraints, we shall draw attention to the design trade-offs made. Examples include the treatment of strings, numbers, the return statement and many more.

1.4 Overview

This section presents a lightning tour of Dart. The goal is to familiarize you with all the core elements of Dart without getting bogged down in detail.

Programming language constructs are often defined in a mutually recursive fashion. It is difficult to present an orderly, sequential explanation of them, because the reader needs to know all of the pieces at once! To avoid this trap, one must first get an approximate idea of the language constructs, and then revisit them in depth. This section provides that approximation.

After reading this section, you should be able to grasp the essence of the many examples that appear in later sections of the book, without having read the entire book beforehand.

Here then, is a simple expression in Dart:

```
3
```

It evaluates, unsurprisingly, to the integer 3. And here are some slightly more involved expressions:

```
3 + 4
(3+4)*6
1 + 2 * 2
1234567890987654321 * 1234567890987654321
```

These evaluate to 7, 42, 5 and 1524157877457704723228166437789971041 respectively. The usual rules of precedence you learned in first grade apply. The last of these examples is perhaps of some interest. Integers in Dart behave like mathematical integers. They are not limited to some maximal value representable in 32 or 64 bits for example. The only limit on their size is the available memory.[1]

Dart supports not just integers but floating-point numbers, strings, Booleans and so on. Many of these built-in types have convenient syntax:

```
3.14159 //  A floating-point number
'a string'
"another string - both double quoted and single quoted forms are supported"
'Hello World' // You've seen that already
true
false //  All the Booleans you'll ever need
[] // an empty list
[0, 1.0, false, 'a', [2, 2.0, true, "b"]] // a list with 5 elements, the last of which is a list
```

As the above examples show, single-line comments are supported in Dart in the standard way; everything after // is ignored, up to the end of the line. The last two lines above show literal lists. The first list is empty; the second has length 5, and its last element is another literal list of length 4.

Lists can be indexed using the operator []

```
[1, 2, 3] [1]
```

The above evaluates to 2; the first element of a list is at index 0, the second at index 1 and so on. Lists have properties length and isEmpty (and many more we won't discuss right now).

```
[1, 2, 3]. length // 3
[].length // 0
[].isEmpty // true
['a'].isEmpty // false
```

One can of course define functions in Dart. We saw our first Dart function, the main() function of "Hello World", earlier. Here it is again

```
main(){
  print('Hello World');
}
```

Execution of a Dart program always begins with a call to a function called main(). A function consists of a header that gives its name and any parameters (our example has

1. Some Dart implementations may not always comply with this requirement. When Dart is translated to Javascript, Javascript numbers are sometimes used to represent integers since Javascript itself does not support an integer datatype. As a result, integers greater than 2^{53} may not be readily available.

none) followed by a body. The body of main() consists of a single statement, which is a call to another function, print() which takes a single argument. The argument in this case is the string literal 'Hello World'. The effect is to print the words "Hello World".

Here is another function:

```
twice(x) => x * 2;
```

Here we declare twice with a parameter x. The function returns x multiplied by 2. We can invoke twice by writing

```
twice(2)
```

which evaluates to 4 as expected. The function twice consists of a signature that gives its name and its formal parameter x, followed by => followed by the function body, which is a single expression. Another, more traditional way to write twice is

```
twice(x) {
  return x * 2;
}
```

The two samples are completely equivalent, but in the second example, the body may consist of zero or more statements—in this case, a single **return** statement that causes the function to compute the value of x*2 and return it to the caller.

As another example, consider

```
max(x, y){ if (x > y) return x; else return y; }
```

which returns the larger of its two arguments. We could write this more concisely as

```
max(x, y) => (x > y) ? x : y;
```

The first form uses an **if** statement, found in almost every programming language in similar form. The second form uses a conditional expression, common throughout the C family of languages. Using an expression allows us to use the short form of function declarations.

A more ambitious function is

```
maxElement(a) {
  var currentMax = a.isEmpty ?
    throw 'Maximal element undefined for empty array' : a[0];
  for (var i = 0; i < a.length; i++) {
    currentMax = max(a[i], currentMax);
  }
  return currentMax;
}
```

The function maxElement takes a list a and returns its largest element. Here we really need the long form of function declaration, because the computation will involve

a number of steps that must be sequenced as a series of statements. This short function will illustrate a number of features of Dart.

The first line of the function body declares a variable named currentMax, and initializes it. Every variable in a Dart program must be explicitly declared. The variable currentMax represents our current estimate of the maximal element of the array.

In many languages, one might choose to initialize currentMax to a known value representing the smallest possible integer, typically denoted by a name like MIN_INT. Mathematically, the idea of "smallest possible integer" is absurd. However, in languages where integers are limited to a fixed size representation defined by the language, it makes sense. As noted above, Dart integers are not bounded in size, so instead we initialize currentMax to the first element of the list. If the list is empty, we can't do that, but then the argument a is invalid; the maximal element of an empty list is undefined. Consequently, we test to see if a is empty. If it is, we raise an exception, otherwise we choose the first element of the list as an initial candidate.

Exceptions are raised using a **throw** expression. The keyword **throw** is followed by another expression that defines the value to be thrown. In Dart, any kind of value can be thrown—it need not be a member of a special Exception type. In this case, we throw a string that describes the problem.

The next line begins a **for** statement that iterates through the list.[2] Every element is compared to currentMax in turn, by calling the max function defined earlier. If the current element is larger than currentMax, we set currentMax to the newly discovered maximal value.

After the loop is done, we are assured that currentMax is the largest element in the list and we return it.

Until now, this tutorial has carefully avoided any mention of terms like object, class or method. Dart allows you to define functions (such as twice, max and maxElement) and variables outside of any class. However, Dart is a thoroughly object-oriented language. All the values we've looked at — numbers, strings, Booleans, lists and even functions themselves are objects in Dart. Each such object is an instance of some class. Operations like length, isEmpty and even the indexing operator [] are all methods on objects.

It is high time we learned how to write a class ourselves. Behold the class Point, representing points in the cartesian plane:

```
class Point {
  var x, y;
  Point(a, b){x = a; y = b;}
}
```

The above is an extremely bare-bones version of Point which we will enrich shortly. A Point has two instance variables (or fields) x and y. We can create instances of Point by invoking its constructor via a **new** expression:

```
var origin = new Point(0, 0);
var aPoint = new Point(3, 4);
```

2. We start at index 0, but we could be slightly more efficient and start at 1 in this case.

```
var anotherPoint = new Point(3, 4);
```

Each of the three lines above allocates a fresh instance of Point, distinct from any other. In particular, aPoint and anotherPoint are different objects. An object has an identity, and that is what distinguishes it from any other object.

Each instance of Point has its own copies of the variables x and y, which can be accessed using the dot notation

```
origin.x // 0
origin.y // 0
aPoint.x // 3
aPoint.y // 4
```

The variables x and y are set by the constructor based on the actual parameters provided via **new**. The pattern of defining a constructor with formal parameters that correspond exactly to the fields of an object, and then setting those fields in the constructor, is very common, so Dart provides a special syntactic sugar for this case:

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

The new version of Point is completely equivalent to the original, but more concise. Let's add some behavior to Point

```
class Point {
  var x, y;
  Point(this.x, this.y);
  scale(factor) => new Point(x * factor, y * factor);
}
```

This version has a method scale that takes a scaling factor factor as an argument and returns a new point, whose coordinates are based on the receiving point's, but scaled by factor.

```
aPoint.scale(2).x  // 6
anotherPoint.scale(10).y // 40
```

Another interesting operation on points is addition

```
class Point {
  var x, y;
  Point(this.x, this.y);
  scale(factor) => new Point(x * factor, y * factor);
  operator +(p) => new Point(x + p.x, y + p.y);
}
```

Now we can write expressions like

```
(aPoint + anotherPoint).y // 8
```

The operator + on points behaves just like an instance method; in fact, it is just an instance method with a strange name and a strange invocation syntax.

Dart also supports static members. We can add a static method inside of Point to compute the distance between two points:

```
static distance(p1, p2) {
  var dx = p1.x - p2.x;
  var dy = p1.y - p2.y;
  return sqrt(dx * dx + dy * dy);
}
```

The modifier **static** means this method is not specific to any instance. It has no access to the instance variables x and y, as those are different for each instance of Point. The method makes use of a library function, sqrt() that computes square roots. You might well ask, where does sqrt() come from? To understand that, we need to explain Dart's concept of modularity.

Dart code is organized into modular units called libraries. Each library defines its own namespace. The namespace includes the names of entities declared in the library. Additional names may be imported from other libraries. Declarations that are available to all Dart programs are defined in the Dart core library which is implicitly imported into all other Dart libraries. However, sqrt() is not one of them. It is defined in a library called dart:math, and if you want to use it, you must import it explicitly.

Here is a complete example of a library with an import, incorporating class Point

```
library points;

import 'dart:math';

class Point {
  var x, y;
  Point(this.x, this.y);
  scale(factor) => new Point(x * factor, y * factor);
  operator +(p) => new Point(x + p.x, y + p.y);
  static distance(p1, p2) {
    var dx = p1.x - p2.x;
    var dy = p1.y - p2.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

We have declared a library called points and imported the library dart:math. It is this import that makes sqrt available inside the points library. Now, any other library that wants to use our Point class can import points.

A key detail to note is that the **import** clause refers to a string 'dart:math'. In general, imports refer to uniform resource indicators (URIs) given via strings. The URIs point the compiler at a location where the desired library may be found. The built-in libraries of Dart are always available via URIs of the form 'dart:σ', where σ denotes a specific library.

There is a lot more to Dart than what we've shown so far, but you should have a general idea of what Dart code looks like and roughly what it means. This background will serve you well as we go into details later in the book.

1.5 Book Structure

The rest of the book is structured around the constructs of the Dart programming language. The next chapter discusses objects, classes and interfaces. These are the core concepts of Dart and are the foundation for all that follows.

Next, we examine libraries in detail, followed by a deeper look at functions. In Chapter 5, we finally take a look at types and the role they play in Dart. We review Dart's expressions and statements in Chapter 6. The final chapters investigate reflection and concurrency.

1.6 Related Work and Influences

The design of Dart has been influenced by earlier languages, in particular Smalltalk[1], Java and Javascript. Dart's syntax follows in the C tradition, via Java and Javascript. Dart's semantics are in some ways closer to Smalltalk - in particular, the insistence on a pure object model.

However, there are crucial differences. Dart introduces its own library-based encapsulation model. This differs from all three of the languages mentioned above. Smalltalk supports object-based encapsulation for fields, with methods and classes universally available. Java has a mix of class-based encapsulation and package privacy, and Javascript relies exclusively on closures for encapsulation.

Like Smalltalk and Java, Dart is class based and supports single inheritance, but it augments this with mixin-based inheritance, very similar to the model first implemented in the Strongtalk dialect of Smalltalk[2]. Because class methods in Dart are not true instance methods as in Smalltalk, but instead Java-style static methods, the models are not exactly the same.

Dart's constructors have a syntactic similarity to those of Java, but in fact differ in critical ways. All of the above topics are discussed in the next chapter.

Dart's approach to type checking is also very close to the one developed for Strongtalk. Types are explored exhaustively in Chapter 5.

Dart's view of concurrency is close to the original actor model (albeit, imperative), again very different from any of the languages cited above. The success of Erlang has been a factor in the adoption of an actor model, yet unlike Erlang, Dart has a non-

blocking concurrency model. Dart also has built-in support for asynchrony heavily influenced by C#. See Chapter 8 for details.

This page intentionally left blank

This page intentionally left blank

Index