Contents

Foreword xi
Preface xv
Acknowledgments xvii
About the Author xix

Chapter 1 Introduction 1
  1.1 Motivation 1
  1.2 Design Principles 2
    1.2.1 Everything Is an Object 2
    1.2.2 Program to an Interface, not an Implementation 2
    1.2.3 Types in the Service of the Programmer 3
  1.3 Constraints 4
  1.4 Overview 4
  1.5 Book Structure 10
  1.6 Related Work and Influences 10

Chapter 2 Objects, Interfaces, Classes and Mixins 13
  2.1 Accessors 14
  2.2 Instance Variables 17
  2.3 Class Variables 17
  2.4 Finals 18
  2.5 Identity and Equality 19
  2.6 Class and Superclass 21
  2.7 Abstract Methods and Classes 22
  2.8 Interfaces 23
  2.9 Life of an Object 24
    2.9.1 Redirecting Constructors 29
    2.9.2 Factories 30
  2.10 noSuchMethod() 30
  2.11 Constant Objects and Fields 31
  2.12 Class Methods 32
  2.13 Instances, Their Classes and Metaclasses 33
## Contents

2.14 Object and Its Methods 34  
2.15 Mixins 36  
2.15.1 Example: The Expression Problem 39  
2.16 Related Work 45  
2.17 Summary 46  

### Chapter 3 Libraries 47  
3.1 The Top Level 47  
3.2 Scripts 48  
3.3 Privacy 48  
3.4 Imports 49  
3.5 Breaking Libraries into Parts 53  
3.6 Exports 55  
3.7 Diamond Imports 56  
3.8 Deferred Loading 57  
3.9 Related Work 58  
3.10 Summary 59  

### Chapter 4 Functions 61  
4.1 Parameters 61  
4.1.1 Positional Parameters 61  
4.1.2 Named Parameters 62  
4.2 Function Bodies 63  
4.3 Function Declarations 64  
4.4 Closures 65  
4.5 Invoking Methods and Functions 66  
4.5.1 Cascades 66  
4.5.2 Assignment 67  
4.5.3 Using Operators 67  
4.6 The Function Class 68  
4.6.1 Emulating Functions 68  
4.7 Functions as Objects 69  
4.8 Generator Functions 71  
4.8.1 Iterators and Iterables 72  
4.8.2 Synchronous Generators 72  
4.9 Related Work 74  
4.10 Summary 74  

### Chapter 5 Types 75  
5.1 Optional Typing 75  
5.2 A Tour of Types 77  
5.3 Interface Types 79  
5.4 Types in Action: The Expression Problem, Typed 82  
5.5 Generics 85  
5.5.1 The Expression Problem with Generics 87
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 vending machine for soda pop where a vending machine for root beer 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:

<table>
<thead>
<tr>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync {... return e; ... }</td>
<td>sync* {... yield e; ... for( ) ... }</td>
</tr>
<tr>
<td>async {... await e ... }</td>
<td>async* {... await e ... yield e; ... }</td>
</tr>
<tr>
<td></td>
<td>async* {... await for( ) ... }</td>
</tr>
</tbody>
</table>

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:

```dart
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 recomputed 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).
1.2 Design Principles

- 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 152415787745770472328166437789971041 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

```dart
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 \(2^{31}\) 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. 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:

```dart
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:

```dart
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

\[(aPoint + anotherPoint).y \div 8\]

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:

```dart
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`

```dart
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
Index

assert, 131
async, 183
async*, 184
await, 183
break, 136
catch, 128
continue, 136
do, 128
dynamic, 78
for, 126
hide
  useful, 55
if, 126
null
  is a message, 176
null
  implicitly returned, 63
rethrow, 129
return, 133
return, 63
super, 120
switch, 129
this, 120
throw, 124
try, 128
while, 128
yield, 134
ClassMirror, 140
Function class, 68, 91
Iterator, 65
Operator class, 72
MirrorSystem, 143
NoSuchMethodError, 30, 62, 77, 78
Object, 78
  API of, 34
    implementation of NoSuchMethod-
Error, 30
Timer, 173, 174
Type class, 33, 46
apply(), 68
basicNew, 152
call(), 69
currentMirrorSystem, 153
double, 79
getName, 143
getSymbol, 143
hashCode, 20
instanceMembers, 140
main(), 59, 64
noSuchMethod(), 30, 70
noSuchMethod, 45
num, 79
reduce(), 65
reflectClass, 140
reflectable, xvii
runtimeType(), 70
runtimeType, 34
spawnUri, 176
staticMembers, 141
toString(), 146
where(), 65

Aarhus, Denmark, xvii
Abstract datatype, 59
accessor(s), 3
actor(s), 1, 10, 185
  in Scala, 185
Ada programming language, 59
Ahé, Peter, xvii
Akka, 185
Anderson, Zachary, xvii
Andersson, Daniel, xvii
Animorphic, xix
Annamalai, Siva, xvii
annotation(s), 102
  processing tool, in Java, 170
  for optimizing reflection, 166
  metadata, 165
ASCII
  identifier(s) must be in, 116
assert statement, 131
AssertionError, 132
assignment, 67
AST, 160
autoboxing
  no such thing, 2
Backus-Naur Form, 155
  Extended, 155
Bak, Lars, xvii
Ben-Gurion
  University, xix
Beta programming language, 107
binding
  user interface, 189
block statement, 125
blocking, 175
boilerplate, 180
Boolean(s), 110
  in tests, 126
  are messages, 176
  in asserts, 132
  in loops, 128
braces
  curly
    delimiting class body or mixin, 36
    delimiting named parameters, 62
Bracha, Teva, xviii
Bracha, Weihong Victoria, xviii
brackets
  square
    delimiting optional parameters, 61
    in indexing operator, 5
break statement, 136, 178
  in switch, 130
builder(s)
  APIs, 67
  mirrors, 189
built-in identifier(s), 119
bytecode
  new, 152
C programming language, 1, 10, 46
  family, 4
C++ programming language, 45, 59, 107
C# programming language, xiii, 11, 59, 107, 169, 185
callback(s), 177, 186
canonicalization
  of constants, 32
capabilities
  for reflection, 166
  mirrors are, 165
  cascade(s), 66
cdspawn, 176
checked mode, 178
  non-Boolean in Boolean expression, 110
class(es), 2, 7, 107
  abstract, 22
  are objects, 33
  as objects, 188
  private, 33
class-based, 1
  encapsulation, 59
  language
    definition, 13
closure(s), 65
equality, 70
  for specifying grammar productions, 156
CLU programming language, 59
Common Lisp programming language, 107
commutativity, 20
compiler error
  ambiguous export of names, 55
  ambiguously imported names, 51
  conflicting prefixes, 52
  Part URI is not a part, 54
  return expression from a generator, 73
Index

URI
- not constant, 54, 55
- interpolated string, 54, 55
- using this in class methods, 32
- violating mixin restrictions, 39
Computational Theologist, xix
concurrency
- shared memory, 175
conditional expression, 125
configuration(s), 188
constant(s), 31, 46
- as default parameter values, 61
- expression(s), 120
- in metadata, 165
- list(s), 114
- map(s), 115
- user-defined objects, 31
constructor(s), 3, 7, 46
- are functions, 64
- classic
- flaws, 45
- redirecting, 29
- support abstraction, 3
continue statement, 136, 178
control constructs, 178
- first class, 74
Cook, William R., 46
Crelier, Régis, xvii

Dart
- designers of, xvii
- VM team, xvii
deadlock, 175
debugging, 170
- fix-and-continue, 169
declaration(s)
- represented by symbols, 113
deferred loading, 57
Denmark, xvi, xvii
dependency-injection, 188, 189
deployment
- and reflection, 166
- benefits from mirrors, 165
- to Javascript, 142
distribution
- benefits from mirrors, 165
do loop, 128
Doench, Greg, xviii
DOM, 70
dot operator, 66
double(s), 110
dynamic language
- definition of, 189
dynamic loading, 169
E programming language, 185
EBNF, 155
ECMA, xvii
capsulation, 59
equality
- of closures, 70
- operator, 19
- user-defined
  - banned in switch, 131
Erlang programming language, 10, 185
Ernst, Erik, xvii
event
- handlers, 174
- loop, 174
example(s), 145
  - Point, 79
exception(s)
- and futures, 173
- in an await, 184
- throwing, 124
- uncaught
  - inside a function, 63
export(s), 55
eexpression(s), 109
  - await, 183, 185
  - conditional, 6, 125
  - throw, 7
factories, 30
field(s)
- constant, 31
Fletch, 169
Flow programming language, xi
fluent API(s), 67
for loop(s), 126
for loops
  asynchronous, 185
for statement, 7
foreign function interface (FFI), 35
function(s), 61, 74
  asynchronous, 183
  emulation of, 68
  expression(s), 115
  generator, 184
  mathematical, 61
type(s), 91
functional programming languages, 74
future(s), 171, 172, 177, 186
  completed with error, 184
  delayed, 174, 185
  in Scala, 185
generator(s), 71, 134, 184
  asynchronous, 184
getter(s), 66
  parameter list of, 61
gradual typing, xi
Hack programming language, xi
Hausner, Matthias, xvii
Heisenbugs, 18
Hello World program, 1
Hewitt, Carl, 185
HTTP, 184
hypertext, 2
IDE(s), 189
  live, 169
  metacircular, 189
identifier(s), 116
identity, 8, 20
  of doubles, 110
  of functions, 69
  of integers, 109
IEEE 754, 109, 110
if statement, 6, 126
import(s), 9, 49
  deferred, 58
  diamond, 56
  weaknesses of, 59
indexing operator, 5
indirection
  any problem in computing can be solved by, 165
inheritance
  mixin-based, 1, 36, 46
  multiple, 36
  of class methods, 33
  of grammars, 156
  single, 36
initializing formal(s)
  types of, 79
integer(s), 5, 109
interface(s), 2, 46, 79, 107
declaration(s)
  unnecessary, 23
introspection, 139
isolate(s), 165, 171, 175, 185
  as objects, 179
Israel, xix
iterable(s), 72, 134
iterator(s), 72
Java programming language, xiii, xix, 10, 59, 107, 169
Javascript
  compilation to, 178
  Javascript programming language, xiii, 5, 10, 46, 165
  broken integers, 109
  compiling Dart
    implications for reflection, 142
JDI, 170
json, 185
JVM, 152
Kronecker, Leopold, 109
label(s), 136
  in breaks, 137
libraries, 9, 47
  as objects, 188
  main method of, 176
  root, 176
Lisp programming language, 46, 169
  Common Lisp dialect, 107
list(s), 113
Index

are messages, 176
listener(s), 70
literal(s), 109
  are constants, 31
  function, 65
live evaluators, 169
live program modification, 170
live programming, 189
loading
dynamic, 188
of libraries, deferred, 57
lock(s), 175
loop(s), 126
Lund, Kasper, xvii

Macnak, Ryan, xvii
main
function, 5
  isolate, 176, 179
  method, 176
map(s), 114
  are messages, 176
McCutchan, John, xvii
Meijer, Erik, xvii, 185
message passing, 171, 175, 186
metacircular
IDE(s), 189
metaclass, 33, 45, 46
metadata, 142, 165, 170
method(s)
  abstract, 22
class, 32
final, 3
main, 176
overloading, 75
static, 32
microtask(s), 174
minification, 142, 168
mirror(s), 139, 170
  API, 143
  builder(s), 189
  system(s)
    in reflectable, 167
Mitrovic, Srdjan, xvii
mixin(s), 36, 46

and expression problem, 39
application, 38
origins of, 46
mobile
application(s), 2
device(s), 1
platform(s), 2
Modula programming language family, 59
Modula-3 programming language, 107
Mountain View, CA, xvii
namespace combinators, 52
namespace(s), 9
NaN
  unequal to itself, 110
new
  bytecode, 152
Newspeak programming language, xix, 46, 170, 185, 188
non-Roman scripts, 116
noSuchMethod, 163, 169, 177, 182
number(s)
  are messages, 176

Oberon programming language, 59
object(s)
  creation of, 121
  everything is an, 2, 13, 46
  inspector(s), 169
object-based encapsulation, 59
object-capability model, 165
object-oriented programming, 68
  pure, 1, 188
  performance of, 190
Odersky, Martin, 91
operator(s), 9, 67, 124
optional typing, xi, xv, 1, 107
  definition of, 3, 75
overriding, 3, 21

parameter(s)
named, 62
optional, 61
  positional, 61
  required, 61
parser(s)
  combinator(s), 155
  using reflectable, 166
part(s), 53
PEG(s), 155
point(s)
  constant, 31
port(s), 175, 182
Posva, Ivan, xvii
prefix(es), 51
  as objects, 188
  required for deferred imports, 58
privacy, 48, 59, 177
promise(s), 177
  in E, 185
property extraction, 122
  and closure equality, 70
proxies, 68, 169, 178
  for isolates, 182
puzzler(s), 119
race(s), 175
Racket programming language, xi, 59, 106
Rasmussen, Linda, xvii
read-eval-print loop(s), 169
receive
  port(s), 175
recursion
  generator definition, 135
  infinite, 132
  mutual
    of grammar productions, 156
    of locals, 125
reference(s)
  forward, 156
reflectable package, 142, 166, 167
reflection, xv, 34, 139, 188
  can violate privacy, 177
  definition of, 139
  implications for speed and size, 142
  used to discover runtime type of an
    integer, 109
reflective change, 169
reflexivity, 20
related work
  asynchrony and concurrency, 185
representation
  independence, 3
representation independence, 45, 66
reserved word(s), 119
rethrow statement, 129
return
  non-local, 74
  statement, 4, 133, 178
    in async function, 184
    inside a generator, 73
root library, 176
Ruby programming language, 46
runtimeType
  of integers, 109
  of strings, 112
Rx, 185
Sandholm, Anders, xvii
Scala programming language, 46, 185
Scheme programming language, 106
scope, of block statement, 125
script(s), 48
security, 169, 177
  benefits from mirrors, 165
  object-capability based, 165
Self programming language, 45, 169
self-modification, 139
send port(s), 175
  are messages, 176
serialization, 179
setter(s), 66, 67
shadowing
  of identifier(s), 118
size
  of deployed reflective code, 168
Smalltalk programming language, 10, 45, 59, 152, 169, 189
smartphone(s), 1
spawning, 176
statement(s), 125
  compound, 126
  grouping, 125
  return, 133
simple, 126
static, 188
    modifier, 9
stream(s), 71, 134, 171, 174, 176, 184, 186
    looping over, 185
    related work, 185
strict function(s), 178
string(s), 111
    are messages, 176
    escape sequences, 112
    implicit concatenation of, 111
    interpolation, 112
    multiline, 111
    raw, 112
Strongtalk, xix, 10, 46, 106, 169
subclassing
    disallowed for bool, 110, 111
    disallowed for double, 110
    switch statement, 129
symbol(s), 113, 168
    and minification, 143
    mapping to/from strings, 143
tables, 1
TC52, xvii
thread(s), 175
tool(s), 188, 189
Torgersen, Mads, 91
trait(s), 46
transitivity, 20
tree-shaking, 142
try statement, 128
turn, 174
Turnidge, Todd, xvii
type(s), 3
    checking
        static, 3
    dynamic checking, 107, 178
    generic, 107
    optional, 187, 190
        See optional typing, 3
    pluggable, 187
    runtime
        of functions, 71
Typescript programming language, xi
UI, 70
unary minus
    creating a symbol
        footnote, 113
uniform reference
    principle of, 46, 66
URI(s), 10, 176, 179
Utah
    university of, xix
value(s)
    default
        of optional parameters, 61
variable(s)
    final
        may not be reassigned, 67
Wadler, Philip, 46
web
    browser(s), 1, 4, 142, 171, 177
    world-wide, 1
while loop, 128
yield statement, 73, 134, 184, 185
Zenger, Matthias, 91