

Dart

В ДЕЙСТВИИ

Крис Бакетт



DMK
ИЗДАТЕЛЬСТВО

MANNING

УДК 004Dart
ББК 32.973-018.2
Б19

Баккет К.

Б19 Dart в действии. – М.: ДМК Пресс, 2013. – 528 с.: ил.

ISBN 978-5-94074-918-9

Dart – язык программирования для разработки веб-приложений, созданный компанией Google. Он обладает современными объектно-ориентированными средствами, как Java или C#, не отказываясь при этом от свойственных JavaScript динамичности и ориентированности на функциональное программирование. Написанные на Dart приложения транслируются в JavaScript, но могут исполняться и непосредственно в браузерах, поддерживающих Dart. В комплекте с Dart поставляются библиотеки и инструментальные средства промышленного качества. На Dart могут быть написаны как клиентская, так и серверная часть приложения, что упрощает процесс разработки.

В этой книге вы познакомитесь с языком Dart и научитесь использовать его для создания приложений, работающих в браузере, – в настольной или мобильной ОС. Это не просто учебное пособие по языку, довольно быстро автор переходит к техническим аспектам работы с Dart. На большинство вопросов, возникающих при чтении, тут же даются ответы!

Издание предназначено веб-программистам разной квалификации, в том числе малознакомым с объектно-ориентированным программированием.

УДК 004Dart
ББК 32.973-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-6172-9086-2 (анг.)

Copyright © 2013 by Manning
Publications Co.

ISBN 978-5-94074-918-9 (рус.)

© Оформление, перевод
ДМК Пресс, 2013



Содержание

Предисловие	14
Вступление	16
Благодарности	18
Об этой книге	20
Об иллюстрации на обложке	25
ЧАСТЬ I. ВВЕДЕНИЕ В DART	26
Глава 1. Здравствуй, Dart	27
1.1. Что такое Dart?	27
1.1.1. Знакомый синтаксис помогает в освоении языка	29
1.1.2. Архитектура одностраничного приложения	30
1.2. Первый взгляд на язык Dart	32
1.2.1. Строковая интерполяция	32
1.2.2. Факультативные типы в действии	34
1.2.3. Традиционная структура на основе классов	36
1.2.4. Определение подразумеваемого интерфейса	38
1.2.5. Фабричные конструкторы для предоставления реализации по умолчанию	39
1.2.6. Библиотеки и область видимости	40
1.2.7. Функции как полноценные объекты	43
1.2.8. Параллелизм с помощью изоляторов	45
1.3. Веб-программирование на языке Dart	46
1.3.1. dart:html: удобная библиотека для работы с моделью DOM браузера	47
1.3.2. Dart и HTML5	48
1.4. Инструментальная экосистема Dart	50
1.4.1. Редактор Dart	50
1.4.2. Виртуальная машина Dart	51
1.4.3. Dartium	51
1.4.4. dart2js: конвертер Dart в JavaScript	51

1.4.5. Управление пакетами с помощью pub	52
1.5. Резюме	53

Глава 2. Программа «Здравствуй, мир» на Dart..... 54

2.1. VM Dart для командных приложений	55
2.2. Программа «Здравствуй, мир» в редакторе Dart	57
2.2.1. Знакомство с инструментами, встроенными в Редактор Dart Editor	58
2.2.2. Dart-скрипты и HTML-файлы	61
2.2.3. Запуск приложения «Здравствуй, мир» в Dartium.....	62
2.2.4. Использование dart2js для конвертации в JavaScript	63
2.2.5. Генерация документации с помощью dartdoc	66
2.2.6. Отладка Dart-кода с помощью точек останова	66
2.3. Импорт библиотек для работы с пользовательским интерфейсом в браузере.....	68
2.3.1. Импорт библиотек Dart.....	69
2.3.2. Доступ к элементам DOM с помощью dart:html	70
2.3.3. Динамическое добавление новых элементов на страницу	71
2.4. Резюме	72

Глава 3. Создание и тестирование Dart-приложения 74

3.1. Конструирование пользовательского интерфейса с помощью dart:html	75
3.1.1. Начальный HTML-файл	76
3.1.2. Создание элементов с помощью dart:html	77
3.1.3. Создание экземпляра Element из фрагмента HTML ...	78
3.1.4. Создание элементов по имени тега	80
3.1.5. Добавление элементов в HTML-документ.....	82
3.2. Добавление интерактивности с помощью событий браузера	86
3.2.1. Добавление предмета в список по нажатии кнопки ...	86
3.2.2. Применение гибкого синтаксиса функций в Dart для обработки событий.....	87
3.2.3. Реагирование на события браузера.....	90
3.2.4. Рефакторинг прослушивателя событий для повторного использования	91

3.2.5. Запрос HTML-элементов в dart:html	92
3.3. Инкапсуляция структуры и функциональности с помощью классов.....	95
3.3.1. Классы в Dart не таят неожиданностей	96
3.3.2. Конструирование класса PackItem.....	97
3.3.3. Инкапсуляция функциональности с помощью методов чтения и установки.....	99
3.4. Автономное тестирование программы.....	103
3.4.1. Создание автономных тестов	105
3.4.2. Определение ожидаемых результатов теста	106
3.4.3. Создание пользовательского сравнителя	107
3.5. Резюме.....	109
ЧАСТЬ II. ЯЗЫК DART	111
Глава 4. Функции и замыкания	112
4.1. Функции в Dart	113
4.1.1. Тип возвращаемого функцией значения и ключевое слово return	116
4.1.2. Передача функции данных с помощью параметров	119
4.2. Функции как полноценные объекты.....	126
4.2.1. Объявления локальных функций.....	128
4.2.2. Определение строгого типа функции	134
4.3. Замыкания.....	137
4.4. Резюме.....	140
Глава 5. Библиотеки и ограничение доступа	142
5.1. Определение и импорт библиотеки	143
5.1.1. Определение библиотеки с помощью ключевого слова library	145
5.1.2. Импорт библиотек.....	147
5.2. Скрытие функциональности путем ограничения доступа к частям библиотеки	155
5.2.1. Ограничение доступа в классах.....	157
5.2.2. Использование закрытых функций в библиотеках....	162
5.3. Организация исходного кода библиотеки	163
5.3.1. Ключевые слова part и part of.....	164
5.4. Упаковка библиотек.....	168

5.5. Скрипты – это исполняемые библиотеки	171
5.6. Резюме	173

Глава 6. Классы и интерфейсы 175

6.1. Определение простого класса	176
6.1.1. Программирование относительно интерфейса класса	177
6.1.2. Формализация интерфейса путем явного его определения	180
6.1.3. Реализация нескольких интерфейсов	181
6.1.4. Объявление аксессуаров свойств	182
6.2. Конструирование классов и интерфейсов	184
6.2.1. Конструирование экземпляров класса	185
6.2.2. Проектирование и использование классов с несколькими конструкторами	187
6.2.3. Использование фабричных конструкторов для создания экземпляров абстрактных классов	187
6.2.4. Применение фабричных конструкторов для повторного использования объектов	190
6.2.5. Использование статических методов и свойств совместно с фабричными конструкторами	191
6.3. Создание константных классов с неизменяемыми полями	194
6.3.1. Финальные значения и свойства	195
6.3.2. Блок инициализации конструктора	195
6.3.3. Использование ключевого слова <code>const</code> для создания константного конструктора	196
6.4. Резюме	197

Глава 7. Расширение классов и интерфейсов 199

7.1. Расширение классов с помощью наследования	200
7.1.1. Наследование класса	201
7.1.2. Наследование конструкторов	203
7.1.3. Переопределение методов и свойств	205
7.1.4. Включение абстрактных классов в иерархию наследования	206
7.2. Все является объектом	210
7.2.1. Проверка отношения «является» <code>Object</code>	210
7.2.2. Использование отношения «является» применительно к <code>Object</code>	212

7.2.3. Использование метода toString(), унаследованного от класса Object	213
7.2.4. Перехват обращений к методу noSuchMethod()	215
7.2.5. Прочая функциональность класса Object	218
7.3. Знакомство с типом dynamic	219
7.3.1. Использование аннотации типа dynamic	220
7.4. Резюме	221

Глава 8. Классы коллекций

8.1. Работа с коллекциями данных	224
8.1.1. Коллекции объектов	226
8.1.2. Использование конкретных реализаций интерфейса Collection	230
8.1.3. Создание специализированных коллекций с помощью обобщенных типов	233
8.1.4. Хранение списков пар ключ–значение в обобщенных словарях	237
8.2. Создание обобщенных классов	242
8.2.1. Определение обобщенного класса	242
8.2.2. Использование своего обобщенного класса	244
8.2.3. Ограничения на параметрические типы	245
8.3. Перегрузка операторов	246
8.3.1. Перегрузка операторов сравнения	247
8.3.2. Неожиданное применение перегрузки операторов	249
8.3.3. Перегрузка операторов доступа по индексу	249
8.4. Резюме	252

Глава 9. Асинхронное программирование с применением обратных вызовов и будущих значений

9.1. Почему веб-приложение должно быть асинхронным	256
9.1.1. Преобразование приложения в асинхронное	259
9.2. Использование обратных вызовов в асинхронном программировании	263
9.2.1. Добавление асинхронных обратных вызовов в Лотерею Dart	265
9.2.2. Ожидание завершения всех асинхронных обратных вызовов перед продолжением	266

9.2.3. Вложенные обратные вызовы как средство управления порядком асинхронного выполнения	269
9.3. Знакомство с типами Future и Completer	271
9.3.1. Передача будущих значений из одного места программы в другое	274
9.3.2. Упорядочение асинхронных вызовов путем сцепления будущих значений	275
9.3.3. Ожидание завершения всех запущенных операций получения будущих значений	276
9.3.4. Преобразование обычных значений в будущее	278
9.4. Автономное тестирование асинхронных API.....	280
9.4.1. Тестирование асинхронных функций обратного вызова	282
9.4.2. Тестирование будущих значений	283
9.5. Резюме	285

ЧАСТЬ III. КЛИЕНТСКИЕ DART-ПРИЛОЖЕНИЯ

Глава 10. Создание веб-приложения на Dart

10.1. Структура одностраничного веб-приложения	289
10.1.1. Приложение DartExpense – постановка задачи	290
10.1.2. Структура Dart-приложения.....	293
10.1.3. Поток выполнения в Dart-приложении	296
10.2. Конструирование пользовательского интерфейса с помощью dart:html	299
10.2.1. Интерфейс Element	299
10.2.2. Конструкторы элементов в действии	303
10.2.3. Организация взаимодействия с представлениями и элементами	305
10.2.4. Построение простой обобщенной сетки	309
10.3. Обработка браузерных событий с помощью dart:html ...	313
10.3.1. Управление порядком прохождения события в браузере	315
10.3.2. Наиболее распространенные типы событий	317
10.4. Резюме	319

Глава 11. Навигация и локальное хранение

данных	321
11.1. Интеграция навигации с браузером	323

11.1.1. Применение функции <code>pushState()</code> для добавления элементов в браузерную историю навигации	324
11.1.2. Подписка на событие <code>popState</code>	326
11.2. Использование куков браузера для повышения удобства работы	329
11.2.1. Сохранение данных в кукe	330
11.2.2. Чтение данных из кука	332
11.3. Локальное хранение данных с помощью Web Storage API	334
11.3.1. Преобразование объектов Dart в формат JSON	335
11.3.2. Преобразование JSON-строк в Dart-объекты	340
11.3.3. Сохранение данных в браузере	341
11.4. Резюме	346

Глава 12. Взаимодействие с другими системами и языками

12.1. Взаимодействие с JavaScript	349
12.1.1. Отправка данных из Dart в JavaScript	352
12.1.2. Получение в JavaScript данных, посланных из Dart	355
12.1.3. Отправка данных из JavaScript в Dart	358
12.2. Взаимодействие с внешними серверами	361
12.2.1. Правило одного домена	363
12.2.2. Использование JSONP для запроса данных у внешнего сервера	364
12.3. Построение допускающих установку браузерных приложений, не требующих сервера	367
12.3.1. Применение технологии AppCache для запуска приложений в автономном режиме	368
12.3.2. Создание пакета приложения, допускающего установку в Chrome	373
12.4. Резюме	377

ЧАСТЬ IV. DART НА СТОРОНЕ СЕРВЕРА

Глава 13. Работа с файлами и протоколом HTTP на сервере

13.1. Запуск серверного Dart-скрипта	381
--	-----

13.1.1. Доступ к аргументам командной строки	384
13.1.2. Доступ к файлам и папкам с помощью dart:io.....	386
13.2. Обслуживание HTTP-запросов от браузера.....	393
13.2.1. Класс <code>HttpServer</code>	395
13.2.2. Передача статических файлов по HTTP.....	397
13.3. REST API для клиентов	399
13.3.1. Отправка содержимого каталога в формате JSON	402
13.3.2. Отправка содержимого файла в формате JSON	403
13.3.3. Добавление пользовательского интерфейса на стороне клиента	404
13.4. Резюме	409

Глава 14. Отправка, синхронизация

и сохранение данных	410
14.1. Передача приложения <code>DartExpense</code> с сервера	411
14.2. Использование веб-сокетов для организации двусторонней связи	411
14.2.1. Соединение через веб-сокеты на стороне клиента	414
14.2.2. Обработка подключения через веб-сокеты на сервере	416
14.2.3. Использование веб-сокетов для межбраузерной синхронизации	419
14.3. Сохранение данных в базе <code>CouchDB</code> с помощью класса <code>HttpClient</code>	426
14.3.1. Краткое введение в <code>CouchDB</code>	427
14.3.2. Совместное использование модельного класса <code>Expense</code> в коде клиента и сервера.....	430
14.3.3. Добавление поддержки сохранения данных на сервере	431
14.4. Резюме	437

Глава 15. Организация параллелизма

с помощью изоляторов	438
15.1. Использование изоляторов как единиц работы	439
15.1.1. Создание изолятора	440
15.1.2. Односторонняя связь с изолятором	442
15.1.3. Двусторонняя связь с изолятором	446

15.2. Динамическая загрузка кода.....	452
15.2.1. Создание изолятора для загружаемого файла.....	454
15.2.2. Определение динамически загружаемого исходного файла	455
15.3. Запуск нескольких исполнителей.....	457
15.4. Резюме	463

Приложение А. Справочное руководство

по языку	464
A.1. Объявления переменных	464
A.1.1. Объявление переменных с ключевым словом <code>var</code> или именем типа.....	465
A.1.2. Объявление финальных (доступных только для чтения) переменных	466
A.1.3. Синтаксис литералов	466
A.1.4. Обобщенные списки и словари	474
A.2. Функции	475
A.2.1. Длинная синтаксическая форма.....	477
A.2.2. Короткая синтаксическая форма.....	477
A.2.3. Параметры функции.....	478
A.2.4. Функции как полноценные объекты	479
A.3. Управление потоком выполнения и итерирование	482
A.3.1. Ветвление потока выполнения	482
A.3.2. Циклы и итерирование	487

Приложение В. Определение классов

и библиотек	491
V.1. Классы и интерфейсы.....	491
V.1.1. Определение классов	491
V.1.2. Наследование классов	502
V.1.3. Абстрактные классы.....	505
V.1.4. Неявные интерфейсы.....	506
V.1.5. Статические свойства и методы.....	508
V.2. Библиотеки и ограничение доступа	509
V.2.1. Определение библиотек	509
V.2.2. Ограничение доступа к элементам библиотеки.....	510

Предметный указатель.....	512
----------------------------------	------------



Глава 1. Здравствуй, Dart

В этой главе

- ❑ Основные сведения о платформе разработки на Dart.
- ❑ Взгляд на язык Dart.
- ❑ Инструментальные средства для разработки Dart-приложений.

Dart – удивительный язык, открывающий возможность создавать сложные веб-приложения быстрее и с меньшим количеством ошибок, чем раньше. В этой главе мы узнаем, как сочетаются язык Dart и его инструментальная экосистема, рассмотрим основные особенности языка и увидим, как его можно использовать для построения одностраничных веб-приложений.

1.1. Что такое Dart?

Dart – структурный язык программирования с открытым исходным кодом, предназначенный для создания сложных веб-приложений, работающих в браузере. Выполнить приложение, написанное на Dart, можно либо в браузере, который поддерживает Dart напрямую, либо предварительно откомпилировав код на JavaScript. Dart обладает знакомым программистам синтаксисом, в нем есть классы и необязательные типы, он является однопоточным. Реализована также модель параллелизма на основе изоляторов, которая допускает параллельное выполнение, – мы обсудим ее в главе 15. Помимо исполнения в браузере и конвертации в JavaScript, написанный на Dart код можно запускать из командной строки в виртуальной машине Dart, что позволяет писать клиентскую и серверную части приложения на одном языке.

Синтаксис Dart очень похож на синтаксис Java, C# и JavaScript. Это не случайно – при создании Dart ставилась задача разработать знакомый язык. Вот крохотный скрипт на Dart, состоящий из единственной функции `main`:

```
main() {
  var d = "Dart";
  String w = "World";
  print("Hello ${d} ${w}");
}
```

- ← Функция main() – точка входа, с которой начинается исполнение полностью загруженного скрипта
- ← Типизация необязательна (тип не указан)
- ← Аннотация типа (задан тип String)
- ← Для вывода на консоль браузера или на stdout производится подстановка в строку

Этот скрипт можно внедрить в HTML-страницу с помощью тега `<script type="application/dart">` и выполнить в браузере Dartium (издание браузера Google Chrome, предназначенное для разработки на Dart). С помощью программы `dart2js` можно также преобразовать его в JavaScript и затем исполнить в любом современном браузере или запустить из командной строки на сервере, воспользовавшись виртуальной машиной Dart (VM Dart).

Но Dart – не просто язык. На рис. 1.1 изображена экосистема инструментов, включающая несколько сред исполнения, редактор, языковые утилиты и полный набор библиотек – все, что нужно для комфортной разработки сложных веб-приложений.

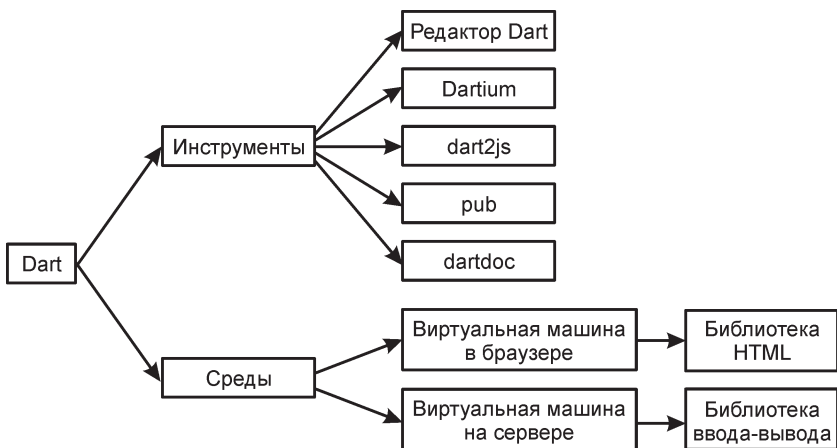


Рис. 1.1. Dart – больше, чем просто язык.
Проект Dart включает целую экосистему

Помимо великолепной инструментальной экосистемы, помогающей при создании приложений, Dart спроектирован так, что покажется знакомым разработчикам как серверных приложений на Java или C#, так и клиентских – на JavaScript или ActionScript.

Основным инструментом разработчика на Dart является браузер Dartium, который позволяет писать и редактировать код и наблюдать за его исполнением. Сочетание Dartium с редактором Dart дополнительно позволяет отлаживать трафик между клиентом и сервером.

1.1.1. Знакомый синтаксис помогает в освоении языка

Одним из важнейших проектных решений было стремление сделать Dart знакомым программистам на JavaScript и на Java/C#. Это помогает быстро освоить новый язык. Если вы владеете хотя бы одним из вышеупомянутых языков, то чтение и понимание написанного на Dart кода не вызовет особых затруднений.

Разработчикам на Java и C# привычны системы типов, классы, наследование и связанные с этим идеи. С другой стороны, на JavaScript пишут как дизайнеры пользовательского интерфейса, которые просто копируют чужой код, чтобы сделать страницу интерактивной (и никогда в жизни не пользовались типами), так и профессиональные программисты, понимающие, что такое замыкание и прототипическое наследование. Чтобы удовлетворить потребности столь разнообразной аудитории, в Dart введена факультативная типизация, позволяющая либо не указывать тип нигде (объявляя переменные с помощью ключевого слова `var`, как в JavaScript), либо везде добавлять аннотации типа (`String`, `int`, `Object` и т. п.), либо применять смешанный подход.

Использование информации о типе позволяет программисту документировать свои намерения, что полезно как для автоматизированных инструментальных средств, так и для коллег. Типичный процесс разработки Dart-приложения подразумевает постепенное построение системы типов по мере того, как программа обретает форму. Добавление или удаление информации о типе не влияет на работу программы, зато позволяет виртуальной машине более эффективно проверять правильность кода. Таким образом, система типов в Dart лежит где-то посередине между динамической типизацией в JavaScript и статической в Java и C#.

В табл. 1.1 сравниваются различные характеристики Dart, Java и JavaScript.

Dart – универсальный язык, позволяющий, подобно JavaScript и Java, создавать приложения разных типов. Но по-настоящему он сверкает при разработке сложных веб-приложений.

Таблица 1.1. Сравнение некоторых характеристик Dart, Java и JavaScript

Характеристика	Dart	Java	JavaScript
Система типов	Факультативная, динамическая	Строгая, статическая	Слабая, динамическая
Функции – полноценные объекты	Да	Можно имитировать с помощью анонимных функций	Да
Замыкания	Да	Да, с помощью анонимных классов	Да
Классы	Да, одиночное наследование	Да, одиночное наследование	Прототипические
Интерфейсы	Да, множественное наследование	Да, множественное наследование	Нет
Параллелизм	Да, на основе изоляторов	Да, на основе потоков	Да, на основе рабочих процессов в HTML5

1.1.2. Архитектура одностраничного приложения

Одностраничные приложения типа Google Mail, Google Instant Search и Google Maps – именно то, для чего предназначен Dart. Исходный код всего приложения (или, по крайней мере, наиболее часто используемых его частей) загружается единственной веб-страницей. Этот – работающий в браузере – код отвечает за построение пользовательского интерфейса и загрузку с сервера отображаемых в нем данных (см. рис. 1.2).

В одностраничных приложениях используется быстрая виртуальная машина на стороне клиента, позволяющая перенести часть обработки с сервера на клиент. В результате сервер способен обслужить больше запросов, так как избавлен от необходимости верстать страницу. Благодаря включенным в Dart библиотекам для работы с HTML, поддерживающим современные HTML5-технологии хранения и кэширования данных в браузере, удастся еще больше повысить производительность приложения и даже дать пользователю работать без подключения к сети.

У любого Dart-скрипта имеется единственная точка входа – функция `main()`, – с которой начинается исполнение скрипта виртуальной машиной Dart. Можно полагаться на то, что весь код, составляющий

приложение, уже загружен к моменту вызова `main`; в Dart невозможно определить и выполнить новую функцию, как в JavaScript не существует ни функции `eval()`, ни какого-либо иного способа модификации исполняемого кода. Именно эта особенность помогает писать на Dart приложения, отвечающие одностраничной архитектуре, потому что есть уверенность, что исполняемый код заранее известен. VM Dart пользуется этой гарантией для ускорения инициализации приложения, применяя мгновенные снимки кучи, – это позволяет загрузить Dart-приложение намного быстрее эквивалентного приложения на JavaScript.

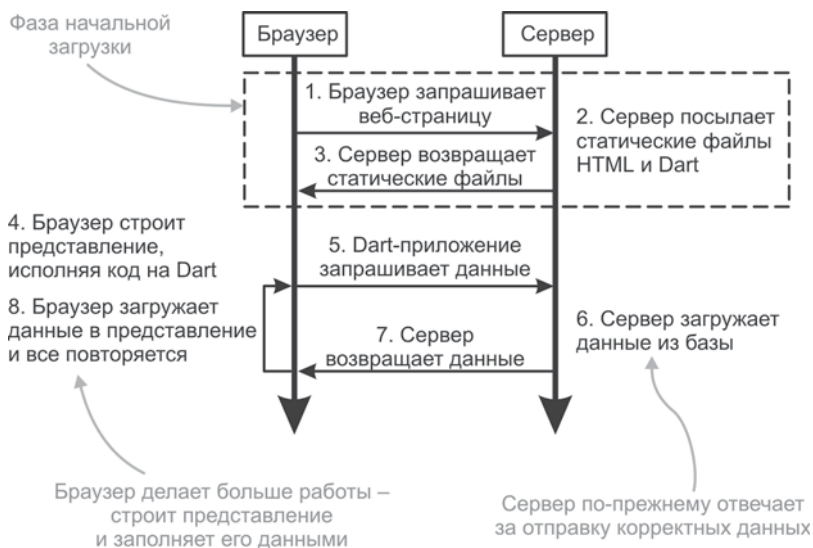


Рис. 1.2. Одностраничное приложение работает в браузере и обращается к серверу только за данными

Памятка

- Dart – язык разработки веб-приложений со знакомым синтаксисом.
- Благодаря наличию инструментальной экосистемы разработка на Dart более продуктивна, чем на эквивалентных динамических языках.
- Факультативная система типов в Dart лежит посередине между динамической типизацией в JavaScript и статической типизацией в Java.
- Аннотации типов могут оказаться очень полезными при коллективной разработке, так как упрощают автоматизированную проверку исходного кода.
- Dart идеален для разработки одностраничных веб-приложений.

Познакомившись с Dart как с платформой разработки, можно приступить к практическому освоению некоторых важнейших особенностей языка.

1.2. Первый взгляд на язык Dart

Dart – полнофункциональный современный язык программирования. Хотя корнями он уходит в Smalltalk, в нем ощущается влияние ряда других языков, в частности Java, C# и JavaScript. В этом разделе мы познакомимся с некоторыми базовыми концепциями языка и кратко упомянем о более сложных механизмах, которые будут подробно рассмотрены впоследствии.

Dart – развивающийся язык

На момент написания этой книги Dart находится на этапе перехода от апробационной технической версии к выпускной версии, которую Google называет «Milestone 1» (Контрольная точка 1). Контрольная точка 1 – это не версия 1, а проведенная на песке черта, от которой можно плясать при разработке и совершенствовании окружающих базовый язык библиотек. Платформа Dart мыслится как полнофункциональная среда разработки типа «все включено», содержащая все необходимое для создания сложных веб-приложений. И компания Google вместе с сообществом Dart сейчас занята разработкой таких библиотек.

Кроме того, контрольная точка 1 определяет состояние, в котором уже можно приступать к созданию приложений в уверенности, что несовместимые изменения в синтаксисе языка будут появляться нечасто. Однако изменения в сопутствующих библиотеках вполне вероятны, поэтому в редактор Dart включена полезная функция наведения порядка (Clean-up), позволяющая применить к коду изменения, внесенные в базовый язык и в библиотеки.

1.2.1. Строковая интерполяция

В веб-приложениях строки используются повсеместно. Dart предлагает несколько способов преобразовать выражение в строку, в том числе функцию `toString()`, встроенную в базовый класс `Object`, и строковую интерполяцию.

Для строковой интерполяции используется знак `$` или выражение `{ }` внутри одиночных или двойных кавычек. Чтобы подставить в строку переменную, необходимо предпослать ей префикс `$`, например: `$name`. Если же требуется подставить результат вычисления выражения, в частности вызова метода, то следует воспользоваться фигурными скобками:

```
"Ответ равен ${5 + 10}"
```

Для создания строки, занимающей несколько строчек, используются три двойные кавычки, а строковые литералы (внутри которых знак \$ не имеет специального смысла) создаются с помощью префикса `r`, например: `r'literal string'`. Для конкатенации двух строк оператор `+` не применяется; необходимо пользоваться либо строковой интерполяцией (`forename $surname`), либо – если речь идет о заранее известных значениях – просто записать две строки одна за другой. Например, выражение

```
var title = "Dart " "в " "действии";
```

порождает строковую переменную со значением "Dart в действии".

В листинге ниже показаны некоторые операции со строками. Для вывода результатов используется встроенная в Dart функция `print`, которая печатает на стандартный выход, если выполняется на сервере, или на отладочную консоль браузера – если в браузере.

Листинг 1.1. Строковая интерполяция в Dart

```
void main() {
  var h = "Hello";
  final w = "World";
  print('$h $w');

  print(r'$h $w');

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

  print("${helloWorld.toUpperCase()}");
  print("Ответ равен ${5 + 10}");

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

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

\$ используется для вычисления простых переменных

← При наличии префикса `r` выводится строковый литерал без интерполяции

Соседние строковые константы конкатенируются

Вычисляемое выражение должно быть заключено в фигурные скобки `{ }`

← При записи многострочных строк первый символ новой строки после `"""` игнорируется

Многострочные строки могут содержать одиночные и двойные кавычки

В случае строковой интерполяции автоматически вызывается функция `toString()`

Эта программа печатает следующее:

```

Hello World
$h $w
Hello World
HELLO WORLD
Ответ равен 15
<div id='greeting'>
  "Hello World"
</div>
Instance of 'Object'
Instance of 'Object'

```

Во время экспериментов с Dart вы часто будете использовать строковую интерполяцию и функцию `print`, чтобы печатать переменные и подставлять значения в HTML-код.

1.2.2. Факультативные типы в действии

Одно из основных отличий JavaScript от Dart – наличие в Dart концепции типов. Однако типы факультативны, и, чтобы воспользоваться строгой типизацией, необходимо включить в код аннотации типов.

Факультативные аннотации типов могут встречаться в объявлениях переменных, параметров функций и возвращаемых значений, а также в определениях классов. Ниже показаны четыре способа объявить строковую переменную `message`. В первых двух аннотации типа нет вообще, а в двух оставшихся есть аннотация `String`, сообщающая программистам и инструментам о том, что в этой переменной будет храниться строковое значение.

```

var messageA;
var messageB = "Hello Dart";

```

Нет аннотаций типов

```

String messageC;
String messageD = "Hello Dart";

```

Аннотация типа присутствует

В двух из четырех случаев переменная `message` инициализируется в момент объявления. Если известно, что значение не изменится после объявления, то следует добавить ключевое слово `final`:

```

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

```

← Использование `final` без аннотации типа
← Использование `final` с аннотацией типа

Подробнее о ключевом слове `final` мы будем говорить ниже.

Чтобы продемонстрировать полезность факультативной типизации, рассмотрим приведенный ниже код, в котором функция `trueIfNull()` принимает два параметра и возвращает `true`, если оба равны `null` (и `false` в противном случае). Сейчас в этом коде нет аннотаций типов, но мы объясним, как ими можно было бы воспользоваться для выражения намерений.

```

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

```

Функция принимает
два значения

```

main() {
  final nums = trueIfNull(1,2); ← Сохраняет "false" в динамической переменной nums
  final strings = trueIfNull("Hello ", null); ← Сохраняет "true" в динамической
                                              переменной strings

  print("$nums");
  print("$strings");
}

```

Выводит переменные
nums и strings на консоль

Поскольку в этом фрагменте аннотаций типов нет, то мы ничего не можем сказать о намерениях разработчика. Имя функции `trueIfNull(a,b)` позволяет предположить, что она принимает два значения типа `int` и возвращает значение типа `bool` (`true` или `false`), но, возможно, разработчик имел в виду что-то другое – например, хотел вернуть строку `"true"` вместо булевого значения. Факультативная типизация в Dart позволяет разработчику документировать свои намерения в виде информации о типах параметрах и возвращаемого значения.

```

bool trueIfNull(int a, int b) { ← Добавлены типы параметров и возвращаемого значения
  return a == null && b == null;
}

```

```

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

```

В объявления переменных
добавлена информация о типе

Примечание. В примере выше встречается тип `bool`. В Dart, в отличие от JavaScript, существует единственный способ представить значение

истинности «ложь», а именно ключевым словом `false`. Нуль и `null` не вычисляются как «ложь».

Добавление аннотаций типа не изменяет работу Dart-приложения, а только сообщает полезную информацию, которой инструменты и ВМ могут воспользоваться для проверки кода и нахождения ошибок типизации. Любая информация о типе может помочь инструментам в проведении статического анализа кода (в редакторе или в командной утилите непрерывной интеграции), а также на этапе выполнения. Программисты, которым в будущем придется сопровождать ваш код, тоже скажут «спасибо».

Совет. Указывайте конкретные типы (например, `String`, `List` и `int`) там, где это полезно для целей документирования, например при объявлении параметров функций, возвращаемых значений и открытых членов класса. Там же, где это не так, например в теле функции, используйте ключевое слово `var` или `final` без аннотации типа. Такой подход рекомендуется в руководстве по стилю программирования на Dart, доступном на сайте www.dartlang.org. Привыкайте к смешанному стилю, поскольку именно так и предполагается писать на Dart.

Факультативная типизация лежит в основе многих механизмов Dart и многократно упоминается на страницах этой книги там, где синтаксические отличия от Java и JavaScript заслуживают пояснения.

Функции как таковые рассматриваются в главе 4.

1.2.3. Традиционная структура на основе классов

В Dart классы и интерфейсы используются традиционным для объектно-ориентированных языков способом, без каких бы то ни было сюрпризов. Поддерживается наследование одному классу и нескольким интерфейсам. Если вы незнакомы с объектно-ориентированным программированием, то, наверное, имеет смысл почитать что-нибудь на эту тему – ресурсов в веб предостаточно. Пока отметим лишь, что объектно-ориентированная модель, принятая в Dart, похожа на то, что есть в Java и C#, и совсем не похожа на JavaScript. Подробнее классы рассматриваются в главах 6 и 7.

По умолчанию все классы в Dart наследуют классу `Object`. У них могут быть открытые и закрытые члены, а удобный синтаксис ме-

тодов чтения и установки позволяет использовать поля и свойства взаимозаменяемым образом, не затрагивая пользователей класса. Ниже приведен пример простого класса.

Листинг 1.2. Простой класс в Dart

```

class Greeter { ← Ключевое слово class определяет новый класс
  var greeting; ← Открытое свойство
  var _name; ← Имена закрытых свойств начинаются с символа _

  sayHello() { ← Открытый метод
    return "$greeting ${this.name}"; ← Строковая интерполяция
  }

  get name => _name; | Синтаксис сокращенной записи методов чтения
  set name(value) => _name = value; | и установки
}

main() {
  var greeter = new Greeter(); ← Ключевое слово new создает новый экземпляр класса Greeter
  greeter.greeting = "Hello "; | Синтаксис присваивания значений полям и свойствам
  greeter.name = "World"; | одинаков
  print(greeter.sayHello());
}

```

В этом простом классе заключено немало интересного. Закрытые члены обозначаются добавлением знака подчеркивания `_` в начало имени. Это принятое в Dart соглашение позволяет читателю кода сразу сказать, что доступ к методу или свойству осуществляется в закрытой области видимости.

Синтаксис методов чтения и установки также полезен, потому что позволяет одинаково работать как с полями, так и со свойствами. Поэтому проектировщик класса может сначала раскрыть поле (как в случае `greeting`), а затем преобразовать его в свойство, к которому можно обращаться только с помощью методов чтения и установки – аксессоров (как в случае `name`); при этом никаких изменений в вызывающий код вносить не придется.

Ключевое слово `this`, которое вызывает столько путаницы в JavaScript, тоже используется традиционным для ООП способом. Оно ссылается на конкретный экземпляр самого класса, а не на владельца класса в данный момент времени (как в JavaScript).

Классы факультативны

В отличие от Java и C#, классы в Dart факультативны. Функции могут находиться в области видимости верхнего уровня, не будучи членами классами. Иными словами, объявлять класс, лишь для того чтобы создать функцию, необязательно. Если выясняется, что класс содержит только служебные методы, то, возможно, заводить его не стоило, а лучше было определить функции на верхнем уровне.

1.2.4. Определение подразумеваемого интерфейса

В Dart, как и в Java и C#, имеются интерфейсы, но для их определения применяется структура класса. Предполагается, что любой класс неявно определяет интерфейс, включающий его открытые члены. В листинге 1.3 определены класс `Welcomer` и функция верхнего уровня `sayHello()`, которая принимает в качестве параметра экземпляр `Welcomer`. Помимо ключевого слова `extends` для реализации наследования в смысле Java и C#, можно также использовать определяемый классом интерфейс с помощью ключевого слова `implements`. Класс `Greeter` реализует открытые методы класса `Welcomer`, что дает возможность использовать его вместо `Welcomer`. Таким образом, программист может принимать во внимание только интерфейс класса, а не его конкретную реализацию.

Листинг 1.3. У любого класса есть неявный интерфейс

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

Классу `Welcomer` можно унаследовать...

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

...но у него имеется также неявный интерфейс, реализуемый классом `Greeter`

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

← Ожидается аргумент типа `Welcomer`

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

```

sayHello(welcomer);

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

```

← Поскольку Greeter реализует интерфейс Welcomer, его можно использовать вместо Welcomer

Способность реализовывать класс, не имеющий явно определенного интерфейса, – весьма полезная особенность Dart. Благодаря ей можно без особого труда создавать подставные классы (mocks) или предоставлять собственную реализацию класса; необязательно явно наследовать общему базовому классу.

1.2.5. Фабричные конструкторы для предоставления реализации по умолчанию

Помимо синтаксиса конструкторов, аналогичного Java и C#, в Dart имеется понятие фабричного конструктора (factory constructor). Можно определить базовый класс, используемый как интерфейс, и включить в него фабричный конструктор, который предоставит конкретный экземпляр по умолчанию. Это особенно полезно, когда в большинстве случаев требуется всего одна реализация некоторого интерфейса.

В листинге 1.4 приведен пример класса IGreetable, в котором имеется фабричный конструктор, возвращающий экземпляр класса Greeter. Класс Greeter реализует интерфейс IGreetable и позволяет пользователям этого интерфейса использовать реализацию Greeter, предоставляемую по умолчанию, не зная о том, что именно они используют. Таким образом, проектировщик класса Greeter может изменить конкретную реализацию, а пользователи интерфейса IGreetable об этом не узнают.

Листинг 1.4. Фабричные конструкторы предоставляют реализации по умолчанию

```

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

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

```

← Определяется интерфейс
← Объявляется метод, который необходимо реализовать

Фабричный конструктор
возвращает
экземпляр Greeter


```

}

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

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

```

Greeter
реализует
интерфейс
IGreetable

Создается экземпляр IGreetable, который
← возвращает реализацию Greeter
← Используется реализация Greeter

Важно отметить, что благодаря этой возможности многие встроенные классы на самом деле являются интерфейсами – в частности, `String` и `int`. У них есть конкретные реализации, возвращаемые фабричными конструкторами. Я буду рассматривать классы, интерфейсы и их взаимодействие с факультативной системой типов во второй части книги.

1.2.6. Библиотеки и область видимости

В Dart имеется возможность организовывать логические наборы исходных файлов. Конечно, можно поместить все Dart-приложение в единственный файл с расширением `.dart`, но это плохо с точки зрения организации кода и навигации по нему. Для решения подобной проблемы в язык Dart включено понятие библиотеки. В Dart библиотекой называется совокупность исходных файлов, которые можно было бы объединить в один файл, но это не сделано, чтобы человеку было проще работать с кодом.

Выше я уже отмечал, что классы в Dart факультативны. Так сделано, чтобы функции можно было определять на верхнем уровне библиотеки. В Dart библиотека представляет собой один или несколько файлов с расширением `.dart`, сгруппированных исходя из каких-то логических соображений; в каждом файле может быть несколько классов (в том числе нуль) и несколько функций верхнего уровня (в том числе нуль). Библиотека Dart может импортировать другие библиотеки, необходимые для работы включенного в нее кода.

Для определения библиотеки служит ключевое слово `library`, для импорта библиотек – ключевое слово `import`, а для ссылки на другие