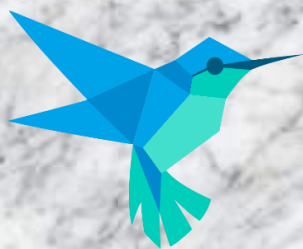


Станислав Чернышев

ОСНОВЫ DART



2021

ОСНОВЫ DART

2021 – v 1.2

Обращение к читателю

Всем добрый день! Меня зовут Станислав Чернышев, я автор данной книги и канала «MADTeacher» на YouTube. Основная моя деятельность — преподавание и выполнение на заказ различных проектов с моими учениками.



За спиной более 10 лет в IT: работа в сфере ВПК, разработка различных аутсорс проектов, управление командой разработчиков, пару выгораний и, само собой, преподавание. Мне нравится обучать, и, чего уж там скрывать — я просто в бешенстве от сложившейся ситуации в сфере образования...

Эта книга — один из маленьких шагов, в попытке изменить чашу весов таким образом, чтобы мои ученики и студенты выходили на рынок труда с актуальным набором компетенций. Если вы не относитесь к их числу, то скорей всего, уже успели начать свою карьеру в IT или только планируете «Захват мира» ^_^ Очень надеюсь, что данная книга поможет вам в этом и предоставит необходимый объем знаний для изучения новой технологии.

Книга предназначена для распространения в открытом доступе. Если у вас имеется желание поблагодарить или поддержать мои начинания, то это можно сделать через пожертвования в группе https://vk.com/mad_enter_it. Все деньги, полученные таким образом, идут на поддержку моей образовательной деятельности и приближают

момент, когда еще раз возьмусь за «перо» для написания новой книги.

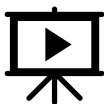
Что касается исходных кодов рассматриваемых в книге примеров, их можно скачать с моего github-репозитория: https://github.com/MADTeacher/dart_basics

Любое копирование текста возможно только с разрешения автора.

© Чернышёв Станислав Андреевич, MADTeacher, 2021



madteacher@bk.ru



[MADTeacher](#)



https://vk.com/mad_enter_it

*Книга посвящается моей жене, Марине, а также
ученикам кооперации «Пафос и превозмогание!!!»,
без чьей помощи и поддержки так и не взялся бы за
её написание*

Содержание

1 Краткая история и встроенные типы данных	10
1.1 Краткая история и основные особенности Dart	10
1.2 Установка и настройка рабочего окружения.....	12
1.3 Встроенные типы данных Dart	17
1.3.1 Числа (int, double).....	17
1.3.2 Строки (String).....	19
1.3.3 Логические значения (bool).....	20
1.3.4 Списки (List).....	20
1.3.5 Множества (Set)	23
1.3.6 Таблицы (Map)	26
1.3.7 Runes и Symbols	28
1.4 Динамический тип данных <i>dynamic</i>	28
1.5 Модификаторы final, const и late	29
1.6 Null-безопасность (Null-safety)	31
Резюме по разделу	35
Вопросы для самопроверки	35
2 Синтаксис, операторы и управляющие конструкции	37
2.1 Основные операторы Dart	37
2.3 Комментарии	42
2.4 Правила именования.....	42
2.5 Управление потоком выполнения кода.....	43
2.5.1 Условный оператор if	43
2.5.2 Операторы циклов (for, for-in, while и do-while)	44
2.5.3 Операторы потока выполнения (break, continue)	47
2.5.4 Оператор выбора потока выполнения (switch-case).....	49
Резюме по разделу	52
Вопросы для самопроверки	52
Упражнения.....	53
3 Функции	54
3.1 Объявление функции	54
3.2 Объявление входных аргументов функции	56
3.3 Необязательные аргументы функции по умолчанию	59
3.4 Область видимости переменных	60
3.5 Обращение к функции через переменную	61
3.6 Функция как входной аргумент другой функции	61
3.7 Анонимные и стрелочные функции	62
3.8 Замыкания.....	64

3.9 Рекурсия.....	66
3.10 Генераторные функции	67
Резюме по разделу	72
Вопросы для самопроверки	72
Упражнения.....	73
4 Библиотеки и пакеты.....	75
4.1 Импортирование кода из файла с расширением «.dart»	76
4.2 Импортирование части функциональности	79
4.2.1 Использование ключевого слова show	79
4.2.2 Использование ключевого слова hide	80
4.3 Отложенная загрузка импортируемого файла или библиотеки.....	81
4.4 Создание и использование библиотеки	81
4.5 Подключение пакета к проекту	84
Резюме по разделу	85
Вопросы для самопроверки	86
Упражнения.....	86
5 Объектно-ориентированное программирование в Dart.....	87
5.1 Объявление класса	89
5.2 Конструктор класса	94
5.2.1 Константный конструктор.....	101
5.2.2 Фабричный конструктор	102
5.3 Статические переменные и методы класса.....	104
5.4 Перегрузка операторов.....	105
5.5 Наследование и переопределение методов.....	107
5.6 Абстрактный класс и интерфейс	111
5.7 Mixins (Примеси)	117
5.8 Generics (Обобщения).....	120
5.9 Enum (Перечисления)	123
Резюме по разделу	124
Вопросы для самопроверки	125
Упражнения.....	125
6 Exceptions (Исключения).....	127
6.1 Конструкция try...catch...finally.....	127
6.2 Генерация исключений и ошибок	130
6.3 Пользовательские исключения	134
6.4 Трассировка стека	134
6.5 Assert (Утверждение).....	135
Резюме по разделу	136
Вопросы для самопроверки	137

Упражнения	137
7 Работа с файлами	138
7.1 Создание, чтение и запись файла	138
7.2 Дополнительные методы для работы с файлами	142
7.3 Работа с JSON-файлами	144
7.2.1 Что за зверь такой JSON?	144
7.2.2 Сохранение данных в JSON-формате.....	146
7.2.3 Загрузка данных из JSON-файла.....	148
7.2.4 Библиотека json_serializable для десериализации и сериализации объектов в JSON-формат	150
Резюме по разделу	158
Вопросы для самопроверки	158
Упражнения	159
8 Асинхронное программирование и Isolate	160
8.1 Event Loop архитектура в Dart.....	161
8.1.1 Базовая концепция цикла и очереди событий	161
8.1.2 Очереди и цикл событий в Dart	162
8.2 Асинхронное программирование	164
8.2.1 Future API.....	165
8.2.2 Ключевые слова async и await.....	171
8.2.3 Stream (Поток).....	173
8.3 Isolate (Изоляты)	181
8.4 Что и когда использовать?	184
Резюме по разделу	184
Вопросы для самопроверки	185
Упражнения	186
9 Сетевое программирование	187
9.1 Передача данных между сервером и клиентом по протоколу TCP	187
9.2 Передача данных между сервером и клиентом по протоколу UDP	191
9.3 HTTP-сервер	193
Резюме по разделу	196
Вопросы для самопроверки	196
Упражнения	196
10 Тестирование	198
10.1 Установка пакета test в проект	199
10.2 Написание тестов	199
10.3 Запуск тестов	202
10.4 Конфигурация тестов.....	204

10.5 Использование пакета mockito для создания фиктивных объектов при проведении тестирования.....	208
Резюме по разделу	213
Вопросы для самопроверки	213
Упражнения.....	213
Список используемых источников.....	214

1 Краткая история и встроенные типы данных

1.1 Краткая история и основные особенности Dart

Dart – объектно-ориентированный язык программирования с сильной статической типизацией и поддержкой обобщенного программирования (шаблоны/дженерики) [1]. Dart не поддерживает множественное наследование, то есть родителем производного класса может выступать только один базовый класс. В тоже самое время, как и в языке программирования Java или C# класс может реализовывать множество интерфейсов. По своему синтаксису Dart очень похож на семейство языков C (Си) – (C++, C#, Java, Kotlin и т. д.).

Dart – молодой язык программирования, который был впервые анонсирован корпорацией Google 10 октября 2011 года. Первая версия языка увидела свет в ноябре 2013 года, а на момент написания книги актуальной является версия 2.12 (Dart 2.12). Начиная с этой версии, Dart поддерживает null-safety (нулевую безопасность, null-безопасность), в основе которой лежат следующие принципы проектирования [2]:

- По умолчанию не допускает значения NULL. Если не указывается явно, что переменная может иметь значение NULL, Dart будет выдавать ошибки на этапе компиляции при присваивании такой переменной значения NULL.
- Использование null-safety позволяет оптимизировать компилятор. Уменьшается не только количество ошибок, связанных с присваиванием NULL, а также объем скомпилированного приложения и повышается скорость его выполнения.

В момент своего появления на свет, Dart позиционировался Google как язык программирования для замены JavaScript, что и сыграло с ним довольно злую шутку. Несмотря на изначальный интерес сообщества программистов к Dart, его не стали повсеместно использовать и чаще всего упоминание об этом языке программирования можно было встретить на форумах при описании ret-проекта. Единственное, что вернуло Dart из того колодца забвения, в который он погружался месяц от месяца – выход первой версии Flutter SDK [3] в конце 2018 года (на момент выхода книги актуальная версия - Flutter 2), где Dart занял место основного языка программирования, на котором посредством Flutter ведется разработка. Сам Flutter представляет собой набор инструментов

для разработки приложений с графическим пользовательским интерфейсом различной сложности как для мобильных устройств, так для Интернета и настольных компьютеров. Это связано с тем, что разработка ведется в рамках одной кодовой базы для следующих платформ: iOS, Android, macOS, Windows, Linux и Web.

Таким образом, сейчас Dart – оптимизированный для клиентской части язык программирования, позволяющий вести разработку быстрых приложений на любой платформе. При этом он дает возможность использовать динамический тип в сочетании с проверками во время выполнения. Это особенно полезно при быстром прототипировании.

В основе технологии, которая используется в компиляторе Dart и позволяет запускать пользовательский код лежит понятие платформа. Всего выделяется 2 вида платформ (см. рис. 1.1):

1. **Native platform.** Данная платформа используется для приложений, разрабатываемых под мобильные устройства и персональные компьютеры с различными типами операционных систем. Сюда входит как виртуальная машина Dart с JIT-компиляцией, так и опережающий компилятор (AOT) для создания машинного кода. Виртуальная машина Dart с JIT-компилятором используется в процессе разработки приложений и предоставляет разработчику возможность горячей перезагрузки приложения (нет необходимости компилировать приложение снова и запускать его), сбор различных метрик в реальном времени и т.д. Когда же приложение готово к развертыванию на целевой платформе или его загрузке в магазин для последующего скачивания пользователем, компилятор Dart AOT обеспечивает опережающую компиляцию в машинный код ARM или x64. Скомпилированный AOT код выполняется внутри среды выполнения Dart, где также присутствует сборщик мусора, в котором применяется подход на основе поколений.

2. **Web platform.** Данная платформа используется для приложений, ориентированных на Интернет. В этом случае также используется 2 вида компиляторов Dart. Первый (dartdevc) используется только в процессе разработки, а второй (dart2js) для окончательной сборки приложения перед его развертыванием. Оба этих компилятора переводят пользовательский код, написанный на Dart в JavaScript.

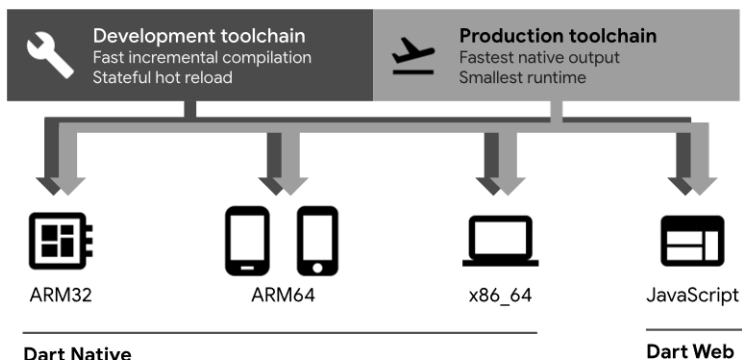


Рисунок 1.1 – Структура платформы языка программирования Dart

Dart **однопоточный язык программирования**, что накладывает ряд ограничений. Да, имеется возможность писать асинхронный код, но «привычного» по другим языкам класса Thread здесь нет. Вместо него используется понятие Isolate. **В отличие от обычного потока Isolate-ы не разделяют общую память, а взаимодействовать между друг другом могут посредством сообщений.**

У Dart **имеется свой менеджер пакетов – pub**, который позволяет устанавливать существующие в хранилище пакеты. В большинстве случаев нет надобности взаимодействовать с ним напрямую. **Достаточно просто прописать в виде зависимости проекта пакет, который необходимо установить, в файл «pubspec.yaml».**

Обязательным требованием к запускаемому приложению является наличие функции верхнего уровня «main», выступающей в роли точки входа (запуска) для разрабатываемого приложения. При этом, аналогично таким языкам программирования, как: C++, C#, Java т.д., каждая команда в коде завершается символом «;».

1.2 Установка и настройка рабочего окружения

Для разработки на языке программирования Dart могут использоваться как различные IDE (IntelliJ IDEA, Eclipse), так и редакторы кода (Visual Studio Code, Vim, Emacs) после установки в них соответствующих плагинов/расширений. Также существует ознакомительный веб-сервис для написания и выполнения Dart кода, располагающийся по адресу: <https://dartpad.dev>.

В рамках первой главы написание кода будет вестись в Visual Studio Code, который можно скачать по следующей ссылке:

<https://code.visualstudio.com/download>

Далее необходимо скачать Flutter SDK под ту операционную систему, которую обычно используете. В моем случае это Windows 10 (рис. 1.2):

<https://flutter.dev/docs/get-started/install>

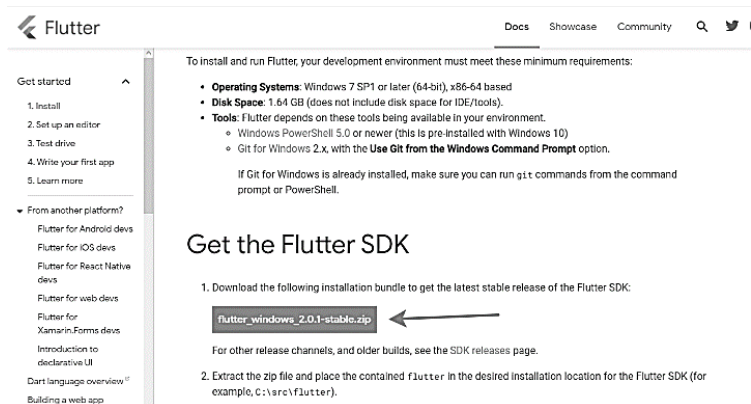


Рисунок 1.2 – Скачиваем архив с текущей стабильной версией Flutter SDK

После того, как архив с Flutter SDK загрузился, распакуйте его в удобную для вас директорию. Обычно этой директорией выступает корневая каталог диска (C, D, F и т. д.). Теперь необходимо прописать путь до распакованного Flutter SDK в переменных средах в переменной «Path», как показано на рисунке ниже:

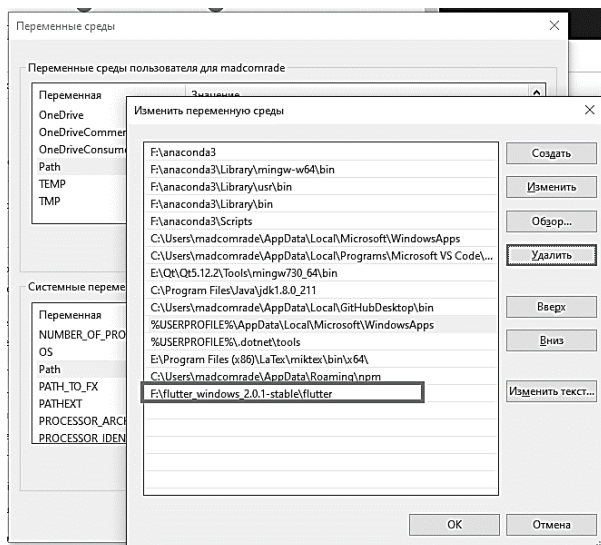


Рисунок 1.3 – Указываем путь до Flutter SDK в переменной «Path»

Запускаем Visual Studio Code, устанавливаем расширение «Dart» и перезапускаем приложение:

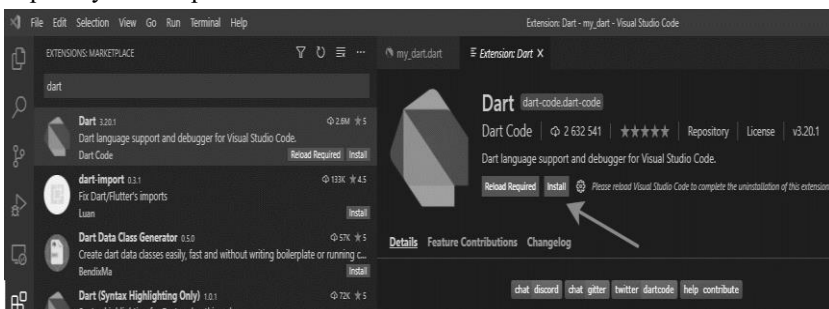


Рисунок 1.4 – Установка расширения «Dart» для VS Code

После перезапуска VS Code создадим новый проект для Dart. Для этого используем английскую связку клавиш «Ctrl+Shift+P» и введем в появившейся командной строке «Dart: New Project». Данная команда может появиться в списке команд до того, как введете её полностью. В этом случае просто выбираем её из списка:

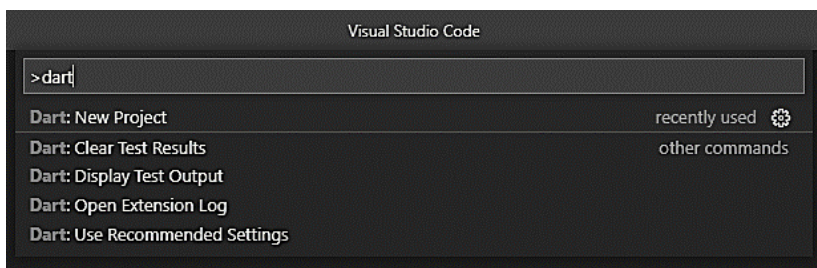


Рисунок 1.5 – Создание нового проекта

На этом шаге обращаем внимание на правый нижний угол приложения. Возможно оно предложит установить какие-то новые зависимости расширения «Dart», что не даст с первой попытки создать новый проект. Если такое произошло, то снова повторяем предыдущий шаг и в появившемся новом списке выбираем «Simple Console Application»:

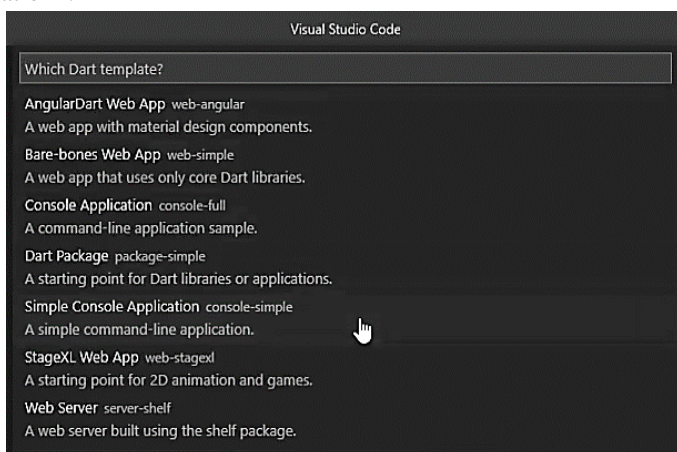


Рисунок 1.6 – Создание консольного приложения

На следующих шагах необходимо выбрать директорию, в которой будет располагаться консольный проект и его имя. После успешно проделанных шагов внешний вид VS Code должен выглядеть примерно следующим образом:

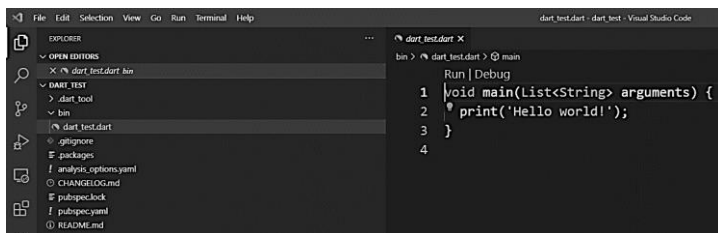


Рисунок 1.7 – Созданный проект

Затем перейдите в файл «pubspec.yaml» и в описании окружения задайте ту версию языка программирования Dart, которую используете. В нашем случае это Dart 2.12. Это действие необходимо потому, что с появлением null-safety еще не все существующие пакеты сделали миграцию своей кодовой базы и использование их с актуальной версией Dart приведет к ошибкам. В соответствии с этим, в создаваемых проектах по умолчанию задается использование предыдущей версии языка программирования:

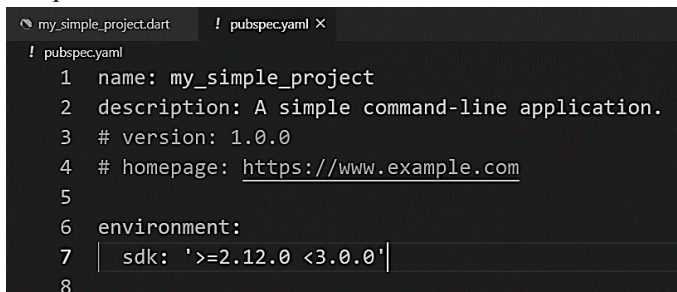


Рисунок 1.8 – Настройка приложения для использования актуальной версии языка программирования Dart

По нажатию на клавишу «F5» можно запустить код в режиме отладки (debug), а воспользовавшись следующим сочетанием клавиш «Ctrl +F5» код запустится без данного режима. Попробуйте один из этих вариантов. В результате должна появиться консоль с текстом «Hello world!».

Теперь VS Code можно полноценно использовать для разработки на языке программирования Dart. Весь последующий код, касающийся условных операторов, циклов и типов данных написан в функции верхнего уровня *main*.

1.3 Встроенные типы данных Dart

Перед знакомством с встроенными типами данных давайте обратимся к документации. В ней говорится, что **все помещаемые в переменную значения являются объектами, которые в свою очередь представляют собой экземпляр класса**. Такая концепция очень похожа на ту, которая применяется в языке программирования Python, в котором всё является объектом. Таким образом, даже числа, строки, функции и null – это объекты.

Существуют следующие встроенные типы данных:

- Числа (int, double);
- Строки (String);
- Логические значения (bool);
- Списки (List);
- Множества (Set);
- Таблицы (Map);
- Руны (Rune);
- Символы (Symbol);
- Значение null (Null).

1.3.1 Числа (int, double)

В Dart всего два числовых типа данных: целочисленные (int) и вещественные, т. е. с плавающей точкой (double).

Целочисленные значения типа int, в зависимости от платформы, могут занимать в памяти не более 64 бит. В виртуальной машине Dart числа типа int могут принимать значения в диапазоне от -2^{63} до $2^{63} - 1$, а при переводе кода в JavaScript используется диапазон значений, который характерен для этого языка программирования: от -2^{53} до $2^{53} - 1$

Числа с плавающей точкой типа double занимают в памяти 64 бита и реализованы в соответствии со стандартом IEEE 754.

Теперь давайте рассмотрим, как можно объявлять переменные числовых типов данных:

```
int a = 5;  
int hex = 0xDEAFF; // 912127  
var b = 10; // int
```

```
double c = 30.5;  
var d = 1.1;  
var exponents = 1.42e5; // 142000.0
```

Ключевое слово `var` перед именем переменной означает, что компилятор Dart сам выведет тип объявляемой переменной в зависимости от того, что разработчик напишет в правой части объявления после символа «`=`».

Как и в других языках программирования со статической типизацией, если мы объявили целочисленную переменную, то компилятор не даст нам записать в эту переменную значение вещественного типа:

```
int a = 5;  
a = 3.5; // error: A value of type 'double' can't be  
         //assigned to a variable of type 'int'.  
var b = 2;  
b = 3.5; // error
```

При этом вещественным переменным мы можем присваивать целочисленные значения:

```
double a = 3.5;  
a = 5;  
  
var b = 2.2;  
b = 3;
```

Типы `int` и `double` являются подтипами типа `num`, в котором в свою очередь определены такие операции с числами, как: `*`, `/`, `+` и `-`. Если мы объявим переменную типа `num` и сначала присвоим ей целочисленное значение, а после вещественное, это не будет считаться ошибкой, так как работа с этими числами осуществляется через экземпляр базового класса типа `num`:

```
num a = 3;  
a = 5.3;
```

1.3.2 Строки (String)

Строки в Dart представляют собой последовательность символов в кодировке UTF-16. Для их объявления (создания) могут использоваться как одинарные, так и двойные кавычки:

```
String s1 = 'Мама мыла раму';  
var s2 = "Мама мыла две рамы";  
var s3 = '''Многострочная  
строка''';
```

Для обращения к конкретному элементу строки по его индексу можно использовать квадратные скобки:

```
print(s2[0]); // М
```

Так как строки — это неизменяемый тип данных (Immutable), то запись вида: `s2[0] = 'П'` приведет к ошибке. В виду этого на основе одного объекта должен быть создан другой, где в процессе создания производятся необходимые изменения:

```
var s4 = 'П' + s2.substring(1); // Пама мыла две рамы
```

В этом случае использовалась операция конкатенация (операция сложения между двумя строками). При этом из строки `s2` были взяты все символы, кроме первого посредством метода `substring`. Если нам необходимо вырезать подстроку определенной длины, то в метод `substring` необходимо передать индекс первого и последнего элемента основной строки. Например:

```
var s3 = 'П' + s2.substring(1, 9); // Пама мыла
```

Узнать длину строки можно, обратившись к атрибуту переменной `length`:

```
print(s2.length); // 18
```

Для перевода всех символов в верхний регистр используется следующий метод:

```
print(s2.toUpperCase()); // МАМА МЫЛА ДВЕ РАМЫ
```

Когда вызываете такие методы у строк, необходимо помнить, что они не влияют на оригинальный объект, а возвращают преобразованное значение, которое необходимо присвоить другой переменной для

последующей работы с ним. Более наглядно это объяснит следующий пример:

```
var s2 = "Мама мыла две рамы";  
s2.toUpperCase();  
print(s2); // Мама мыла две рамы  
var s3 = s2.toUpperCase();  
print(s3); // МАМА МЫЛА ДВЕ РАМЫ
```

Теперь рассмотрим ситуацию, которая будет встречаться довольно часто при написании кода. Это перевод числа в строку и наоборот:

```
// String -> int  
var myInt = int.parse('34'); // строка в число
```

```
// String -> double  
var myDouble = double.parse('11.45');
```

```
// int -> String  
String s1 = 14.toString();  
String s2 = myInt.toString();
```

```
// double -> String  
String s3 = 3.14159.toStringAsFixed(2); // 2 числа после точки  
String s4 = myDouble.toString();
```

Более подробно с методами, которые предоставляет класс `String` можно ознакомиться в официальной документации Dart [4].

1.3.3 Логические значения (bool)

Переменные типа `bool` могут принимать только 2 значения: `true` и `false`. Их объявление производится следующим образом:

```
bool a = false;  
var b = true;
```

1.3.4 Списки (List)

Списки – позиционно упорядоченные коллекции объектов. В отличие от строк списки можно модифицировать на месте путем присваивания по индексу или вызовом некоторых списковых методов. Это позволяет использовать списки как довольно гибкий инструмент для представления коллекций объектов, таких как: перечня продуктов в чеке, текущие дела на день и т.д.

Так как в Dart нет такого типа переменных, как массивы, вместо них используются списки. В виду этого они делятся на два типа:

- с фиксированным количеством элементов;
- с произвольным количеством элементов.

По умолчанию создается список с произвольным количеством элементов. Аналогично числам и строкам списки можно объявлять несколькими способами. Отдав вывод типа объектов, с которыми работает список на откуп Dart, либо задать явно:

```
var myList1 = [ 1, 2, 3];  
List<int> myList2 = [1, 2, 3];  
var myList1 = <int>[]; //пустой список
```

Изменение значения элемента списка производится следующим образом:

```
myList1[0] = 20;  
print(myList1); // [20, 2, 3]
```

Так как списки по умолчанию могут хранить только объекты одного типа данных, то если добавить или изменить значение элемента списка значением другого типа данных, это приведет к ошибке:

```
myList1[0] = 20.4; // error: A value of type 'double' can't  
                  // be assigned to a variable of type 'int'.  
myList1.add('Oo'); // error: A value of type 'String' can't  
                  // be assigned to a variable of type 'int'.
```

Для создания списка из неизменяемых (константных) элементов необходимо использовать ключевое слово **const**:

```
var constList = const [4, 2, 1];  
constList[0] = 5; // exception: Cannot modify an unmodifiable list
```

Далее приведем некоторые возможные операции со списком:

```
var myList = <int>[]; // пустой список для элементов типа int  
myList.add(4);  
// добавляем элементы списка в myList  
myList.addAll([1, 3, 5]);  
print(myList); // [4, 1, 3, 5]  
print(myList.length); // 4
```

```
// удаляем элемент из списка, хранящийся по индексу 0
```

```

myList.removeAt(0);
print(myList); // [1, 3, 5]
// проверяем наличие 3-ки в списке
print(myList.contains(3)); // true

// создаем новый список, добавляя в него элементы
существующего
var myList2 = <int>[1, ...my_list];
print(myList2); // [1, 1, 3, 5]

```

Чтобы создать список с фиксированным количеством элементов (массив) можно воспользоваться следующим способом его объявления:

```

var myList = List<int>.filled(5, 0);
print(myList); // [0, 0, 0, 0, 0]
myList[0] = 200;
print(myList); // [200, 0, 0, 0, 0]
myList.add(10); // exception: Cannot add to a fixed-length list

```

Как и в Python, в Dart поддерживаются довольно гибкие способы формирования списков:

```

var check = false;
var myList = [
  'Привет!',
  'Это список!',
  'Из 3-х эл-ов!',
  if (check) 'Ой! Уже из 4-х!'];
print(myList);
// [Привет!, Это список!, Из 3-х эл-ов!] при check = false
// [Привет!, Это список!, Из 3-х эл-ов!, Ой! Уже из 4-х!] при check = true

```

Либо списки могут быть сформированы так:

```

var intList = [1, 2, 3, 4, 5, 6, 7, 8];
var stringList = [
  for (var i in intList) '#$i'
];
print(stringList); // [#1, #2, #3, #4, #5, #6, #7, #8]
// формируем новый список только из тех элементов,
// что делятся на 2 без остатка
var newList = [
  for (var i in intList) if (i % 2 == 0) i
];

```

```
print(newList); // [2, 4, 6, 8]
```

Более подробно с методами, которые предоставляет класс List можно ознакомиться в официальной документации Dart [5].

1.3.5 Множества (Set)

Множество – неупорядоченная совокупность объектов одного типа данных, в которой не может быть дубликатов. Их часто используются для двух целей: удаление дубликатов и проверка принадлежности. Как и другие типы множество можно объявить явным и неявным образом:

```
var mySet = <int>{1, 2, 5, 5, 5, 6, 7, 8};  
// Set<int> mySet = {1, 2, 5, 5, 5, 6, 7, 8};  
// Set<int> mySet = {}; // пустое множество  
// var mySet = <int>{}; // пустое множество  
print(mySet); // {1, 2, 5, 6, 7, 8}
```

Для примера давайте рассмотрим ситуацию, в которой у нас имеется список из различных чисел и нам необходимо отбросить все существующие в нем дубликаты для формирования нового списка:

```
var myList = <int>[1, 1, 1, 2, 2, 5, 5, 5, 6, 7, 8];  
print(myList); // [1, 1, 1, 2, 2, 5, 5, 5, 6, 7, 8]  
var newList = Set<int>.from(myList).toList();  
print(newList); // [1, 2, 5, 6, 7, 8]
```

Аналогичный подход можно применить, если хотим на основе строки получить не дублируемый список символов, из которых она состоит:

```
var str = 'Мама мыла раму';  
print(str); // Мама мыла раму  
var newList = Set<String>.from(str.toLowerCase().split('')).toList();  
print(newList); // [м, а, , ы, л, р, у]
```

Как и в случае со строками, значение элементов множества не может быть изменено:

```
var mySet = <int>{1, 2, 5, 5, 5, 6, 7, 8};  
mySet[0] = 3;  
// The operator '[' isn't defined for the type 'Set<int>'.
```

Добавляются и удаляются элементы в множество следующим способом:

```
var mySet = <int>{1, 2, 5, 5, 5, 6, 7, 8};  
mySet.add(10);  
print(mySet); // {1, 2, 5, 6, 7, 8, 10}  
mySet.remove(1);  
print(mySet); // {2, 5, 6, 7, 8, 10}
```

Эти методы мы использовали ранее для добавления и удаления элементов из списка. При этом, аналогично списку, можно довольно гибко создавать новые множества:

```
var myList = <int>[1, 2, 5, 5, 5, 6, 7, 8];  
var newSet= <int>{  
  for (var i in myList) if (i % 2 == 0) i  
};  
print(newSet); // {2, 6, 8}
```

Dart поддерживает всего три математические операции над множествами:

- Объединение (union);
- Разница (difference);
- Пересечение (intersection).

Давайте рассмотрим, как эти операции реализуются в коде:

```
var mySetA = <int>{1, 2, 5, 6, 7, 8};  
var mySetB = <int>{20, 22, 5, 6, 73, 88, 25};  
print(mySetA.union(mySetB));  
// {1, 2, 5, 6, 7, 8, 20, 22, 73, 88, 25} объединение  
print(mySetA.difference(mySetB)); // {1, 2, 7, 8} A-B  
print(mySetB.difference(mySetA)); // {20, 22, 73, 88, 25} B-A  
print(mySetA.intersection(mySetB)); // {5, 6} пересечение A и B
```

В каждом приведенном случае в процессе вызова метода у множества, создается новый экземпляр класса множества, в который и записывается результат операции, после чего он возвращается вызываемым методом. То есть исходное множество остается без изменений:

```
var mySetA = <int>{1, 2, 5, 6, 7, 8};  
var mySetB = <int>{20, 22, 5, 6, 73, 88, 25};  
var newSet = mySetA.union(mySetB);
```



```
print(newSet); // {1, 2, 5, 6, 7, 8, 20, 22, 73, 88, 25}
print(mySetA); // {1, 2, 5, 6, 7, 8}
print(mySetB); // {20, 22, 5, 6, 73, 88, 25}
```

Для проверки вхождения элемента в множество следует использовать метод `contains`:

```
var mySetA = <int>{1, 2, 5, 6, 7, 8};
print(mySetA.contains(2)); // true
print(mySetA.contains(10)); // false
```

К работе с такими типами данных, как `List`, `Set`, `Map`, пользовательские классы и т. д., необходимо подходить с осторожностью. Это связано с тем, что при объявлении новой переменной и инициализацией её посредством присваивания существующей переменной в системе, копируется не значение, а ссылка на этот объект:

```
void main(List<String> arguments) {
  var myList = [10, 20];
  var newList = myList;
  newList.add(40);
  add(myList, 20);
  print(myList); // [10, 20, 40, 20]
  print(newList); // [10, 20, 40, 20]
}
```

Из приведенного примера видно, что переменной `newList` присвоилась ссылка на список, с которым была связана переменная `myList`. В итоге получилось, что через две переменные мы работаем с одним и тем же объектом. Для того, чтобы работать с копией списка, то есть с новым объектом, который будет проинициализирован значениями существующего объекта, Dart предоставляет следующие механизмы:

```
void main(List<String> arguments) {
  var myList = [10, 3, 4, 1];
  var newList = List.from(myList);
  newList.add(2);
  print('Элементы myList: $myList');
  // Элементы myList: [10, 3, 4, 1]
  print('Элементы newList: $newList');
  // Элементы newList: [10, 3, 4, 1, 2]
```

```

var newList1 = [...myList];
newList1.add(77);
print('Элементы myList: $myList');
// Элементы myList: [10, 3, 4, 1]
print('Элементы newList1: $newList1');
// Элементы newList: [10, 3, 4, 1, 77]
var newList2 = [...myList];
newList2.add(5);
print('Элементы myList: $myList');
// Элементы myList: [10, 3, 4, 1]
print('Элементы newList2: $newList2');
// Элементы newList: [10, 3, 4, 1, 5]
}

```

Но если элементы списка относятся к типу данных, на которые копируются ссылки, то следует предельно осторожно копировать такие объекты, так как изменение элементов создаваемого объекта будут сказываться на существующем объекте:

```

void main(List<String> arguments) {
var myList = [[10, 3, 4, 1]];
var newList = List.from(myList);
newList[0].add(77);
print('Элементы myList: $myList');
// Элементы myList: [[10, 3, 4, 1, 77]]
print('Элементы newList: $newList');
// Элементы newList: [[10, 3, 4, 1, 77]]
}

```

Подобные механизмы копирования имеются у множеств и таблиц (класс Map).

1.3.6 Таблицы (Map)

Так как не принято переводить название этого типа данных, то будем придерживаться этой традиции. Map представляет из себя объект, который связывает ключи и значения. Как ключи, так и значения могут быть объектами любого типа данных. Ключ по своей сути уникален и не может встречаться несколько раз, в то время как значение может использоваться сколько угодно раз и связываться с различными

ключами. Говоря простыми словами **Map** представляет собой серию пар «ключ:значение».

Ниже приведен пример объявления этого типа данных:

```
var myMap = <String, String>{
    //ключ //значение
    'first': 'Мама',
    'second': 'мыла',
    'fifth': 'раму'
};

print(myMap); // {first: Мама, second: мыла, fifth: раму}

var myMap2 = <int, String>{
    1: 'Мама',
    2: 'мыла',
    3: 'раму'
};
print(myMap2); // {1: Мама, 2: мыла, 3: раму}

var newMap = Map<String, int>(); // пустой объект
var newMap = <String, int>{};
```

Изменение значения, которое хранится по ключу или добавление новой пары «ключ:значение» производится практически идентично:

```
var myMap = <int, String>{
    1: 'Мама',
    2: 'мыла',
    3: 'раму'
};
myMap[1] = 'Бабушка';
// добавляем новую пару «ключ:значение»
myMap [10] = 'по утрам!';
print(myMap);
// {1: Бабушка, 2: мыла, 3: раму, 10: по утрам!}
```

Давайте приведем некоторые примеры организации работы с **Map**:

```
var myMap = <int, String>{
    1: 'a', 2: 'b', 3: 'c',
    4: 'd', 5: 'e', 6: 'f'
};
```

```
// количество элементов
print(myMap.length); // 6
// список ключей
print(myMap.keys.toList()); // [1, 2, 3, 4, 5, 6]
// список значений
print(myMap.values.toList()); // [a, b, c, d, e, f]
// удалить пару «ключ:значение»
myMap.remove(1); // указываем ключ
print(myMap); // {2: b, 3: c, 4: d, 5: e, 6: f}
// удалить все связки пар, которые подходят
// под условие, что значение ключа не делится
// на 2 без остатка
myMap.removeWhere((key, value) => (key % 2 != 0));
print(myMap); // {2: b, 4: d, 6: f}
```

1.3.7 Runes и Symbols

Runes аналогичны строкам с тем отличием, что они представляют собой последовательность символов в кодировке UTF-32, а не UTF-16. В этом случае каждый символ представляет собой запись вида `'\uXXXX'`, где `XXXX` четырех числовое значение в шестнадцатеричной системе счисления. Например, букве «П» соответствует представление `'\u041F'`.

Объекты типа **Symbol** представляют собой некоторые идентификаторы для ссылки на различные элементы API, такие как библиотеки или классы. Они применяются не так часто и возможно вам никогда не придется их использовать. Для того, чтобы объявить **Symbol** необходимо использовать `«#»`:

```
var mySymbol = #myAPI;
print(mySymbol); // Symbol("myAPI")
```

1.4 Динамический тип данных *dynamic*

Так как Dart позиционировался как замена JavaScript, то без наличия такого типа данных как **dynamic** было бы даже нереально попытаться сказать об этом вслух. **Dynamic** лежит в основе всех объектов Dart, что позволяет разработчику, в случае необходимости, одной переменной присваивать значения совершенно различных типов данных:

```
dynamic myValue = 3;
```

```

myValue = 4.10;
print(myValue); // 4.10
myValue = 'oO';
print(myValue); // oO
myValue = [3, 4, 'w'];
print(myValue); // [3, 4, w]

```

В тех случаях, когда имеется необходимость хранить в списке элементы произвольных типов, в качестве типа элементов списка задается `dynamic`:

```

List<dynamic> myList = [0, 4, 4.5, 'str'];
// var myList = <dynamic>[0, 4, 4.5, 'str']
myList[0] = 20.4;
myList.add('Oo');
print(myList); // [20.4, 4, 4.5, str, Oo]

```

Аналогичный подход можно применять и с типом данных *Map*. Так, например, если нам необходимо иметь ключи и значения из различных типов данных, то *Map* необходимо объявить следующим способом:

```

var myMap = <dynamic, dynamic>{
  1: 'Мама',
  'Oo': 10,
  3: [10, 4, 5]
};
print(myMap); // {1: Мама, Oo: 10, 3: [10, 4, 5]}

```

Несмотря на наличие такого гибкого механизма его не рекомендуется использовать повсеместно, так как это может повлечь за собой трудно отлавливаемые ошибки не только в самом коде, но и в логике разрабатываемого приложения.

1.5 Модификаторы `final`, `const` и `late`

Модификаторы `final` и `const` по своей сути очень похожи. Переменные, впереди типа которых ставятся эти модификаторы не могут изменяться в процессе выполнения программы. Ключевое их отличие заключается в том, что константные переменные должны быть инициализированы в момент объявления, а переменные с

модификатором `final` можно инициализировать позже, но только один раз:

```
final int a;  
const int b = 10;  
a = 4; // ok  
b = 5; // error: Constant variables can't be assigned a value.  
a = 3; // error: The final variable 'a' can only be set once.
```

Тип, в случае использования данных модификаторов можно явно не указывать, он будет выведен Dart автоматически:

```
final a;  
const b = 10;  
const name = 'Петр';
```

Ключевое слово `const` можно использовать не только для объявления неизменяемых переменных. Его еще используют для создания постоянных значений и объявления конструкторов, посредством которых создаются неизменяемые значения. Благодаря этому любая переменная может иметь неизменяемое значение:

```
// нельзя добавлять элементы средством метода add  
var first = const [];  
final second = const [];  
const third = [];  
  
first = const [10, 3, 23]; // ok  
second = const [10, 3, 23]; // error  
third = const [10, 3, 23]; // error  
first[0] = 30;  
// Unsupported operation: Cannot modify an unmodifiable list
```

Модификатор `late` был добавлен в версии Dart 2.12 и имеет два варианта использования:

1. Для объявления переменной, не хранящей значение `null`, инициализация которой происходит уже после ее объявления;
2. Для ленивой инициализации переменной.

Отличие `final` от `late` заключается в том, что переменные с модификатором `final` не могут быть объявлены на верхнем уровне кода. При этом, переменные, объявленные с одним из этих модификаторов, должны быть проинициализированы до их использования. Иначе в процессе выполнения приложение выбросит исключение:

```
late String name;
// final int variable; // ошибка

void main(List<String> arguments) {
  final int variable; // ok
  // print(name);
  // LateInitializationError: Field 'name' has not been initialized
  name = 'Михаил';
  print(name); // ok
}
```

1.6 Null-безопасность (Null-safety)

Чтобы рассмотреть тему null-безопасности нам придется забежать немного вперед, но это позволит более подробно объяснить те моменты, где она используется и почему была введена в версии Dart 2.12.

Основная проблема, когда у нас объект может хранить значение `null` связана с тем, что это может вызвать падение программы и увеличение кодовой базы проекта за счет введения дополнительных проверок на `null`. Переменная экземпляра класса имеет некоторое состояние и реализует поведение, в тоже самое время, если переменная хранит ссылку на `null` мы не можем реализовать необходимое поведение в рамках приложения. `Null` ничего не знает о поведении объекта, в связи с чем, когда мы пытаемся вызвать какой-либо метод у переменной, происходит падение приложения.

Начиная с версии Dart 2.12 все объявляемые переменные создаются как `null-safety`, то есть переменной объявляемого типа данных нельзя присвоить значение `null`. Также, если мы не проинициализировали переменную до её использования, компилятор выведет ошибку:

```
class Cat {
  void helloMaster(){
    print("Мяу-у-у!!!");
  }
}

void main(List<String> arguments) {
  Cat myCat;
  myCat.helloMaster(); // The non-nullable local variable 'myCat'
                      // must be assigned before it can be used.
}
```

```
}
```

Давайте рассмотрим следующую ситуацию. Вы живете в квартире с кошкой. Каждый раз, когда открываете холодильник, она начинает неистово мяукать, пытаясь надавить на жалость, чтобы её покормили колбасой. Весь код этой ситуации можно написать следующим образом:

```
class Cat {  
    void helloMaster(){  
        print("Мяу-у-у-у!!!");  
    }  
}  
  
void openFridge(Cat cat){  
    cat.helloMaster(); // Мяу-у-у-у!!!  
}  
  
void main(List<String> arguments) {  
    var myCat = Cat();  
    openFridge(myCat);  
}
```

В данном случае в коде подразумевается, что кошка всегда есть в квартире. А вдруг её забрал кто-то из родственников к ветеринару или она настолько любит гулять, что раз в день ваша вторая половинка выходит с ней на улицу и как раз в этот момент вы решили открыть холодильник? Так вот, открывая холодильник вы всё равно услышите это протяжное «**Мяу-у-у-у!!!**». Как так? Кошки же не должно быть в квартире, а значит переменная не должна хранить ссылку на экземпляр класса. Она должна указывать на `null`.

С учетом `null-safety` нельзя экземпляру класса кошки, который объявлен в примере выше, присвоить значение `null`:

```
void main(List<String> arguments) {  
    var myCat = Cat();  
    myCat = null; // error: A value of type 'Null' can't be assigned  
                  // to a variable of type 'Cat'.  
}
```

В этом случае нам необходимо явно указать компилятору, что объявляемая переменная не является `null-safety`, то есть она может

ссылаться на `null`. Для этого используется символ «?» сразу после объявления типа переменной:

```
void main(List<String> arguments) {
  Cat firstCat = null; // error: A value of type 'Null' can't
                        //be assigned to a variable of type
                        'Cat'.

  Cat? myCat = null;
}
```

Так как теперь у нас кошка может то присутствовать, то отсутствовать в квартире, то в метод `openFridge` необходимо передавать аргумент типа «`Cat?`» и учитывать ссылается передаваемый аргумент на `null` или на экземпляр класса. Для этого Dart предоставляет несколько возможностей, таких как операторы «?.», «??» и «!». Оператор «?.» вызовет метод экземпляра класса, если переменная не ссылается на `null`, иначе никакой метод вызываться не будет:

```
void openFridge(Cat? cat){
  cat?.helloMaster();
}

void main(List<String> arguments) {
  Cat? myCat;
  Cat? newCat = Cat();
  openFridge(myCat); // ничего не выведется
  openFridge(newCat); // Мяу-у-у-у!!!
}
```

Оператор «??» позволяет организовать заглушку, если переменная ссылается на `null`. Иначе работа будет производиться с экземпляром класса, переданным в функцию. Но его лучше использовать с осторожностью:

```
void openFridge(Cat? cat){
  final someCat = cat ?? Cat();
  someCat.helloMaster();
}

void main(List<String> arguments) {
  Cat? myCat;
  Cat? newCat = Cat();
}
```

```

    openFridge(myCat); // Мяу-у-у-у!!!
    openFridge(newCat); // Мяу-у-у-у!!!
}

```

То есть при использовании этого оператора, даже если кошки нет в квартире, по комнате все равно пронесется протяжное «Мяу-у-у-у!!!».

Используя последний оператор «!». вы говорите компилятору, что хоть переменная и может ссылаться на `null` вы более чем уверены, что она ссылается на экземпляр класса:

```

void openFridge(Cat? cat){
    cat!.helloMaster();
}

void main(List<String> arguments) {
    Cat? newCat = Cat();
    openFridge(newCat); // Мяу-у-у-у!!!
}

```

В тоже самое время, при передаче в функцию `null` приложение выбросит исключение:

```

void openFridge(Cat? cat){
    cat!.helloMaster(); // _CastError (Null check operator used on a null value)
}

void main(List<String> arguments) {
    openFridge(null);
}

```

Таким образом, использование не `null`-safety переменных оправдано только в том случае, если по другому логику работы программы не реализовать.

Дополнительно обращайтесь внимание на то, что не во всех случаях значения не `null`-safety переменных могут присваиваться `null`-safety переменным:

```

void main(List<String> arguments) {
    Cat? cat;
    Cat newCat = cat; // error: A value of type 'Cat?' can't
                      // be assigned to a variable of type 'Cat'.
}

```

```
void main(List<String> arguments) {
  Cat? cat = Cat();
  Cat newCat = cat; // ok
}
```

Аналогичным образом можно объявлять и не null-safety переменные встроенных типов данных:

```
int? a;
String? name = null;
// и т.д.
```

Резюме по разделу

В первом разделе помимо краткой истории языка программирования Dart, были рассмотрены его основные особенности, встроенные типы данных и способы работы с ними. Приведенные способы работы со списками, словарями и т. д., постоянно будут встречаться в ходе работы над реальными проектами, так что лучше их изучить в самом начале пути и не заглядывать каждый раз в справочник для уточнения по методам и способам работы с ними.

Отдельно стоит отметить концепцию null-safety, что позволяет не беспокоиться о наличии значения `null` в переменных и предоставляет разработчикам механизм введения в код разрабатываемого приложения не null-safety типов данных и операторы для работы с ними.

Вопросы для самопроверки

1. Для замены какого языка программирования разрабатывался Dart?
2. Какие 2 платформы используются в компиляторе Dart? Для чего они используются и какие между ними различия?
3. Какие ключевые особенности у языка программирования Dart?
4. Какие встроенные типы данных предоставляет Dart?
5. Какому числовому типу данных можно присваивать как целочисленные, так и вещественные значения?
6. В чем отличие типа `String` от `Rune`?
7. Что такое список? Как использовать список в Dart в качестве массива?
8. Что такое множество? Приведите его ключевые особенности.

9. Какой тип данных необходимо использовать, если необходимо объявляемой переменной присваивать значения различных типов данных?
10. В чем схожи, а чем отличаются модификаторы `final` и `const`?
11. В чем схожи, а чем отличаются модификаторы `final` и `late`?
12. Перечислите ключевые моменты концепции null-безопасности (`null-safety`)?

2 Синтаксис, операторы и управляющие конструкции

2.1 Основные операторы Dart

Операторы в Dart классифицируются следующим образом:

- арифметические операторы;
- операторы сравнения;
- операторы проверки;
- операторы присваивания;
- логические операторы;
- побитовые операторы;
- условные выражения;
- каскадная запись;
- другие операторы.

Арифметические операторы представлены в таблице ниже. К ним также относятся как префиксные, так и постфиксные операторы инкремента и декремента значения переменной:

Таблица 2.1 – Арифметические операторы

Оператор	Описание	Примеры
+	Сложение	$10 + 5 = 15$ или $10 + -3 = 7$
-	Вычитание	$15 - 5 = 10$ $25 - -3 = 28$ $11.98 - 7 = 4.98$
*	Умножение	$2 * 2 = 4$ $7 * 3.2 = 22.4$ $-2 * 4 = -8$
/	Деление	$12 / 4 = 3$ или $7 / 3 = 2.334$
%	Деление по модулю	$4 \% 2 = 0$ $9 \% 2 = 1$ $13.2 \% 4 = 1.199$
~/	Целочисленное деление	$17 ~/ 5 = 3$ $10 ~/ 3 = 3$
++var	Префиксный инкремент	<code>var = var + 1;</code>
var++	Постфиксный инкремент	<code>var = var + 1;</code>
--var	Префиксный декремент	<code>var = var - 1;</code>
var--	Постфиксный декремент	<code>var = var - 1;</code>

Далее рассмотрим операторы сравнения:

Таблица 2.2 – Операторы сравнения

Оператор	Описание	Примеры
<code>==</code>	Проверка на равенство	<code>1 == 1 (true)</code> <code>true == false (false)</code> <code>"test" == "test" (true)</code>
<code>!=</code>	Проверка на неравенство	<code>1 != 2 (true)</code> <code>false != false (false)</code> <code>"test" != "Test" (true)</code>
<code>></code>	Проверка на то, что значение левого операнда больше правого	<code>3 > 2 (true)</code> <code>2 > 3 (false)</code>
<code><</code>	Проверка на то, что значение левого операнда меньше правого	<code>3 < 2 (false)</code> <code>2 < 3 (true)</code>
<code>>=</code>	Проверка на то, что значение левого операнда больше или равно правого	<code>3 >= 1 (true)</code> <code>3 >= 3 (true)</code>
<code><=</code>	Проверка на то, что значение левого операнда меньше или равно правого	<code>5 <= 5 (true)</code> <code>-4 <= -21 (false)</code>

В следующей таблице представлены операторы проверки, которые удобно использовать для проверки типов во время выполнения кода:

Таблица 2.3 – Операторы проверки

Оператор	Описание	Примеры
<code>as</code>	Данный оператор используется для приведения одного типа данных к другому	
<code>is</code>	true, если объект имеет указанный тип	<code>double a = 3.4;</code> <code>print(a is double); // true</code> <code>print(a is String); // false</code>
<code>is!</code>	true, если объект не имеет указанный тип	<code>double a = 3.4;</code> <code>print(a is! double); // false</code> <code>print(a is! String); // true</code>

Перечень существующих операторов присваивания в Dart представлен в таблице 2.4:

Таблица 2.4 – Операторы присваивания

Оператор	Описание	Примеры
=	Оператор присваивания	var a = 3; var a = 2.3;
+=	a +=b, что равносильно a = a+b;	var a = 3; a +=2; // 5
-=	a -=b, что равносильно a = a-b;	var a = 3; a -= -2; // 5
*=	a *=b, что равносильно a = a*b;	var a = 3; a *=2; // 6
/=	a /=b, что равносильно a = a/b;	var a = 9; a /=3; // 3
%=	a %=b, что равносильно a = a%b;	var a = 9; a %=3; // 0
~/=	a ~/=b, что равносильно a = a~/b;	var a = 9; a ~/= 3; // 3
>>=	a >>=b, что равносильно a = a>>b;	var a = 8; a >>= 2; // 2
<<=	a <<=b, что равносильно a = a<<b;	var a = 8; a <<= 2; // 32
^=	a ^=b, что равносильно a = a^b;	var a = 7; a ^= 2; // 5
&=	a &=b, что равносильно a = a&b;	var a = 7; a &= 2; // 2
=	a =b, что равносильно a = a b;	var a = 7; a = 2; // 7

Для рассмотрения работы побитовых операторов необходимо ввести два значения a и b. Примем a = 33 в десятичной системе счисления, что эквивалентно 0010 0001 в двоичной системе счисления и b = 87 (0101 0111):

Таблица 2.5 – Побитовые операторы

Оператор	Описание	Примеры
&	Побитовое логическое «И» между операндами	a & b = 0000 0001 (1)
	Побитовое логическое «ИЛИ» между операндами	a b = 0111 0111 (119)
^	Побитовое логическое «исключающее ИЛИ» между операндами	a ^ b = 0111 0110 (118)
~	Логическое отрицание. Инвертирует значения бит операнда к которому применяется	~a = 1101 1110 (-34)

Оператор	Описание	Примеры
<<	Побитовый сдвиг влево. Применяется к одному операнду. Эквивалентно умножению на 2^n , где n – число на которое производится сдвиг	$a \ll 2 = 1000\ 0100$ ($33 * 2^2 = 132$)
>>	Побитовый сдвиг вправо. Применяется к одному операнду. Эквивалентно целочисленному делению на 2^n	$a \gg 2 = 0000\ 0100$ ($33 \sim / 2^2 = 8$)

К логическим операторам относят обычные логические операции, как «И», «ИЛИ» и «НЕ»:

Таблица 2.6 – Логические операторы

Оператор	Описание	Примеры
&&	Логическое «И» между операндами	true and true = true Во всех остальных случаях false
	Логическое «ИЛИ» между операндами	false and false = false Во всех остальных случаях true
!	Логическое отрицание	!false = true !true = false

В следующей таблице приведены существующие в Dart условные выражения:

Таблица 2.7 – Условные выражения

Выражение	Описание
condition ? expr1 : expr2	Если условие истинно, то вычисляется и возвращается expr1, в противном случае вычисляется и возвращается значение expr2
expr1 ?? expr2	Если expr1 не равно null, возвращается его значение, иначе вычисляется и возвращается значение expr2

Операторы, которые не вошли в перечисленную классификацию операторов Dart относятся к категории «другие операторы»:

Таблица 2.8 – Другие операторы

Оператор	Описание
()	Используется для вызова функций
[]	Ссылается на значение в списке в соответствии с задаваемым индексом
.	Позволяет обратиться к свойствам объекта
?.	Как и оператор «.», только дополнительно выполняет проверку на null. В том случае если объект хранит значение null обращения к его свойству не производится и не выбрасываются никакие исключения

Последний вид операторов, который предоставляет Dart – это каскадные операторы (.., ?..), которые позволяют выполнять последовательность операций над одним и тем же объектом:

```
class Cat {
  late final int old;
  late final String name;
  set old(int old) {
    this.old = old;
  }

  set name(String name) {
    this.name = name;
  }

  void helloMaster(){
    print("Мя-у-у-у!!!");
  }
}

void main(List<String> arguments) {
  var cat = Cat()
  ..name = 'Мяся'
  ..old = 4
  ..helloMaster();
  // аналогично записи ниже
  var newCat = Cat();
  newCat.name = 'Мяся';
  newCat.old = 4;
  newCat.helloMaster();
}
```

2.2 Комментарии

Комментарии в Dart делятся на 2 типа: однострочные и многострочные. В первом случае используется «//», после чего идет комментарий, который не переносится на следующую строку:

```
// комментарий  
var a = 10; // еще один комментарий
```

Если комментарий будет занимать более 2-х строк, то его каждую строку необходимо начинать либо с «//», либо использовать многострочный формат записи комментария:

```
/*  
Сверх  
длинный комментарий  
*/  
var a = 10;
```

Комментарии можно использовать не только для того, чтобы комментировать происходящее в коде. Например, посредством комментариев можно исключить выполнение определённой строки или блока кода (то есть закомментировать их). Но сильно этим увлекаться не стоит, так как такой подход засоряет чистоту вашей кодовой базы, из-за чего в последующем обязательно возникнут трудности у новых людей в команде.

Строки же документации, которые позволяют использовать инструмент *dartdoc* для автоматической генерации документации вашего проекта начинаются с «///». Считается плохим тоном в тех частях, где пояснения должны попасть в документацию использовать простые комментарии, так как в этом случае они будут пропущены. Более подробно с тем, как принято документировать код в проектах, разрабатываемых с использованием Dart, можно ознакомиться в руководстве по документированию проектов, расположенному на официальном сайте [6].

2.3 Правила именования

При написании кода на Dart лучше придерживаться следующих рекомендаций при объявлении переменных, функций, классов и их методов:

1. При объявлении переменных, функций и методов классов используется верблюжий стиль, а само название начинается с маленькой буквы (*lowerCamelCase*). Для логического разделения слов в объявляемой переменной необходимо использовать символ в верхнем регистре: *myCatName*. Имя же объявляемого класса начинается с большой буквы (*UpperCamelCase*): *DailySchedule*;
2. Нельзя использовать в начале объявляемого имени числовые значения;
3. Регистр символов имеет значение. Так, например, *var CHECK = 10;* и *var check = 10;* две совершенно разные переменные;
4. Не используйте в качестве имен ключевые слова Dart;
5. Если имя переменной, функции и т.д. начинается с символа «_», то она является приватной (для импортирующего код модуля).

2.4 Управление потоком выполнения кода

Для управления потоком выполнения кода в Dart используются следующие виды операторов:

- Условный оператор *if*;
- Операторы циклов (*for*, *for-in*, *while* и *do-while*);
- Операторы потока выполнения (*break*, *continue*);
- Оператор выбора потока выполнения (*switch-case*).

2.4.1 Условный оператор *if*

Оператор *if*, с необязательным оператором *else*, выбирает действия, которые будут выполняться в процессе работы программы в зависимости от условий, которые проверяются в скобках, после объявления оператора *if*. Общая форма конструкции «если-то-иначе» представлена ниже:

```
if (условие1){
    блок1
}
else if (условие2){
    блок 2
}
...
else if (условие(n-1)){
    блок (n-1)
```

```

}
else{
    блок (n)
}

```

Условие представляет собой проверку на истинность и если в результате вычисления при проверке условия возвращается `true`, то будет выполнен код из блока 1, иначе будет выполняться проверка в блоках `else if`. В том случае, если не выполнилось ни одно условие, то выполнится код из блока `else`:

```

void main(List<String> arguments) {
    var a = 10;
    var b = 30;
    var c = 7;
    if (a > b) {
        print('a > b');
    } else if (a > c) {
        print('a > c'); // <- a > c
    }
    else{
        print('Ни то и ни другое');
    }
}

```

2.4.2 Операторы циклов (for, for-in, while и do-while)

Цикл (оператор) `for` позволяет выполнить блок кода определенное количество раз. В общем виде структуру этого цикла можно представить следующим образом:

```

for (действие до начала цикла;
    условие выхода из цикла;
    действие по завершению текущего шага цикла) {
    // блок кода
}

```

Любой из элементов (действие или условие выхода) при объявлении цикла может быть не задан. Так, например, следующий цикл является «бесконечным», поскольку при его объявлении не указано условие завершение цикла:

```

for(;;){

```

```
}
```

В следующем коде цикл `for` выполнится 5 раз, после чего напечатается строка:

```
var str = '';
for(var i = 0; i <= 4; i++){
    str += i.toString();
}
print(str); // 01234
```

Этот код может быть переписан следующим образом:

```
var str = '';
var i = 0;
for(; i <= 4;){
    str += i.toString();
    i++;
}
print(str); // 01234
```

По сути что один, что другой цикл выполняет одинаковые действия. Всё отличие заключается в чистоте и читаемости кода. В следующем примере давайте используем цикл `for` для заполнения списка:

```
var myList = <int>[];
for(var i = 0; i <= 4; i++){
    myList.add(i);
}
print(myList); // [0, 1, 2, 3, 4]
```

Цикл `for-in` позволяет перебирать значения, возвращаемые любым объектом, поддерживающим итерацию: списки, множества и т. д., то есть объект должен представлять собой коллекцию из элементов:

```
var myList = <int>[for (var i = 0; i <= 3; i++) i];
for (var it in myList){
    print(it); // 0 1 2 3
}
var mySet = <int>{1, 2, 5, 6, 7, 8};
for (var it in mySet){
    print(it); // 1 2 5 6 7 8
}
```

Переменные с типом `Map<K,V>` не могут использоваться совместно с циклом `for-in`. Для прохода по всем элементам любых коллекций можно использовать метод `forEach()`:

```
var myMap = <int, String>{
    1: 'Мама',
    2: 'мыла',
    3: 'раму'
};
myMap.forEach((key, value) {
    print('$key => $value');
});
1 => Мама
2 => мыла
3 => раму
```

Для того, чтобы пройти по всем элементам строки используйте следующий вид записи цикла:

```
var myStr = 'Hi!';
for(var i = 0; i < myStr.length; i++){
    print(myStr[i]); // H i !
}
```

Либо можно применить связку таких методов, как `split()` и `forEach()`:

```
var myStr = 'Hi!';
myStr.split('').forEach((element) {
    print(element); // H i !
});
```

Принципы работы циклов `while` и `do-while` довольно похожи. Ключевое различие заключается в том, что цикл `while` может ни разу не выполниться. Это связано с тем, что сначала проверяется условие его выполнения. В том случае, если оно возвращает значение `false`, поток выполнения кода переходит к командам и операторам, расположенным за циклом. Цикл `do-while` выполнится хотя бы один раз, после чего уже идет проверка условия: выполнить цикл по новой или выйти из цикла.

Структуру этих циклов можно представить следующим образом:

```
while (условие выхода из цикла) {
    // блок кода
}
```

```
do{
    // блок кода
}
while (условие выхода из цикла);
```

Давайте приведем пример, как можно использовать цикл **while** вместо цикла **for**:

```
var myStr = 'Hi!';
var i = 0;
while(i < myStr.length){
    print(myStr[i]); // H i !
    i++;
}
```

```
i = 0;
do{
    print(i); // 0 1 2
    i++;
}while(i < 3);
```

2.4.3 Операторы потока выполнения (**break**, **continue**)

Оператор **continue** используется для немедленного перехода в начало цикла, в котором он был вызван:

```
var i = 13;
while(i > 0){
    i--;
    if (i % 2 == 0){
        continue;
    }
    print(i); // 11 9 7 5 3 1
}
```

Оператор **break** используется для немедленного выхода из цикла, в котором он был вызван. Представим ситуацию, что у нас имеется вложенный цикл (цикл в цикле). При использовании оператора **break** внутри вложенного цикла, поток управления перейдет к циклу верхнего

уровня, который продолжит выполняться. Ниже приведен пример выхода из бесконечного цикла, посредством оператора **break**:

```
var i = 33;
while(true){
    if (i <= 3){
        break;
    }
    i--;
}
print(i); // 3
```

При использовании меток («название_метки:») с оператором **break** можно сразу выйти из нескольких вложенных друг в друга циклов:

```
void main(List<String> arguments) {
    mainLoop: for(var i = 0; i < 3; i++){ // метка «mainLoop:»
        print('start main loop');
        for(var x = 0; x < 3; x++){
            print('start second loop');
            for(var y = 0; y < 3; y++){
                print('start external loop');
                if (y >= 1){
                    print('break external loop');
                    break mainLoop;
                }
            }
            print('end external loop');
        }
        print('end second loop');
    }
    print('end main loop');
}
print('end loops');
}

start main loop
start second loop
start external loop
end external loop
start external loop
break external loop
end loops
```


2.4.4 Оператор выбора потока выполнения (switch-case)

Оператор `switch` позволяет сравнивать целочисленные, строковые переменные или константы времени компиляции с помощью оператора сравнения «`==`». Подаваемые на вход оператора `switch` и сравниваемые объекты должны быть экземплярами одного и того же класса. При этом данный класс не должен переопределять оператор «`==`». Хорошей практикой является использовать перечисления (Enum) при работе с этим операторами.

Рассмотрим пример работы `switch-case`:

```
var command = 'close'; // проверяемое значение
switch (command) {
  case 'close': // если значение в command == 'close'
    print('closed'); // <- closed
    break;
  case 'open': // если значение в command == 'open'
    print('open');
    break;
  default: // если не подошел ни один вариант
    print('default');
}
```

Проверка на соответствие значения, подаваемого на вход оператора `switch` происходит в блоках `case`. Если в конце каждого блока `case` используется оператор `break`, то выполнится только один блок, после чего управление перейдет к коду, следующему за `switch-case`. Если этот оператор отсутствует и в теле текущего блока `case` нет кода, то выполниться код из следующего блока `case` и так до тех пор, пока на пути его выполнения не встретится оператор `break` или не выполнится полностью оставшаяся часть `switch-case`.

Для начала рассмотрим пример, что будет если мы уберем один из операторов `break` в предыдущем коде:

```
var command = 'close';
switch (command) {
  case 'close':
    print('closed');
    // error: The 'case' should not complete normally.
  case 'open':
    print('open');
```

```

    break;
  default:
    print('default');
}

```

Так как мы хотим, чтобы между открытием и закрытием не было разницы, немного модифицируем этот пример:

```

var command = 'close';
switch (command) {
  case 'close':
  case 'open':
    print('open/close'); // <- open/close
    break;
  default:
    print('default');
}

```

Если нам необходимо после завершения одного из блоков **case** перейти к выполнению другого, то есть ввести некоторый аналог машины состояния, можно использовать оператор **continue** и метку («название_метки:»):

```

var command = 'open';
switch (command) {
  prepare:
  case 'prepare':
    print('prepare'); // 2 <- prepare
    break;
  case 'close':
    print('closed');
    continue prepare;
  case 'open':
    print('open'); // 1 <- open
    continue prepare;
  default:
    print('default');
}

```

Необязательный блок **default** выполнится в том случае, когда не подойдет ни одно из значений из объявленных блоков **case**:

```

var command = 'Oo';

```

```

switch (command) {
    prepare:
    case 'prepare':
        print('prepare');
        break;
    case 'close':
        print('closed');
        continue prepare;
    case 'open':
        print('open');
        continue prepare;
    default:
        print('default'); // <- default
}

```

Помимо операторов `break` и `continue` блок `case` может завершаться операторами `return` или `throw`.

Теперь давайте забежим немного вперед и перепишем последний пример кода с использованием перечислений:

```
enum DoorCommand {none, open, close, prepare}
```

```

void main(List<String> arguments) {
    var command = DoorCommand.open;
    switch (command) {
        prepare:
        case DoorCommand.prepare:
            print('prepare'); // 2 <- prepare
            break;
        case DoorCommand.close:
            print('closed');
            continue prepare;
        case DoorCommand.open:
            print('open'); // 1 <- open
            continue prepare;
        default:
            print('default');
    }
}

```

Резюме по разделу

В данном разделе мы рассмотрели существующие операторы и базовые синтаксические конструкции языка программирования Dart: условный оператор, циклы, оператор выбора потока выполнения. Они будут постоянно встречаться вам в процессе написания приложений, поэтому понимание их принципов работы снизит вероятность ошибок в логике разрабатываемого программного продукта.

Также нами было рассмотрены правила наименования переменных. Что касается того, с заглавной или строчной буквы будет начинаться имя переменной, функции, класса и т.д., эти правила носят рекомендательный характер. Это совершенно не значит, что вы обязаны их придерживаться, но само следование этим правилам является «хорошим тоном» при написании приложений. Такое положение дел связано с тем, что его придерживается огромное количество программистов, в соответствии с чем код становится более читаемый. Согласитесь, куда приятнее вникать в то, что написано в коде, когда все следует одному и тому же соглашению по наименованию.

Вопросы для самопроверки

1. Какие операторы существуют в Dart?
2. На сколько типов делятся комментарии в Dart? Приведите их примеры.
3. Какие правила при наименовании переменных существуют?
4. Является ли блок **else** обязательным при использовании условного оператора **if**?
5. Для чего используются циклы?
6. Какие операторы циклов существуют в Dart?
7. Чем отличается цикл **for** от **for-in**?
8. Чем отличается цикл **while** от **do-while**?
9. Для чего используются операторы **break** и **continue**?
10. Для чего используется оператор выбора потока выполнения **switch-case**?
11. Какие возможности предоставляют метки при их использовании в операторе выбора потока выполнения **switch-case**?
12. Подаваемые на вход оператора **switch** и сравниваемые объекты должны быть экземплярами одного и того же класса?

13. Обязательно ли наличие при окончании блока **case** оператора **break**? Если нет, то в каких случаях?
14. Какой цикл необходимо использовать для итерации по объектам типа **Map<K, V>**?

Упражнения

1. Посчитайте двумя способами вхождение каждого символа в строке «Представим ситуацию, что у нас имеется вложенный цикл (цикл в цикле).».
2. Выведите в терминал числа в диапазоне от 23 до 35 используя различные циклы.
3. Заполните список целочисленными значениями от 0 до 99, используя циклы и механизм гибкого формирования списков.
4. На основе сформированного в предыдущем задании списка сформируйте новый список и добавьте только те элементы из существующего, которые нацело делятся на 5. Используйте для этого циклы и механизм гибкого формирования списков.
5. Используя циклы выведите числа в диапазоне от -35 до -1 с шагом 1, 4 и 7.
6. Найдите максимальное и минимальное по значению число в переменных **a**, **b**, **c**, используя условный оператор **if**.
7. Выведите элементы списка [4, 5, 6, 7, 2, 1, 2, 3, 4] в обратной последовательности.
8. Выведите все элементы списка [4, 5, 6, 7, 2, 1, 2, 3, 4] в терминал, кроме тех, что содержат значение 2 и 6.
9. Посчитайте сумму элементов списка [4, 5, 6, 7, 2, 1, 2, 3, 4].
10. Найдите среднеарифметическое значение элементов списка [4, 5, 6, 7, 30, 22, 2, 39, 41].
11. Удалите из списка [1, 3, 4, 1, 4, 5, 7, 3, 8, 5] повторяющиеся значения.
12. Посредством оператора **switch-case** выведите в терминал уведомление является подаваемая на вход буква гласной или согласной.
13. Выведите элементы списка [4, 5, 6, 7, 2, 1, 2, 3, 4] с их номером индекса.
14. Используя оператора **switch-case** выведите в терминал уведомление том, какое значение из диапазона от 0 до 5 подается на его вход.

3 Функции

Поскольку Dart является объектно-ориентированным языком программирования, то и функции в нем – это объекты. Хотя это и не указывается явно, у них есть тип – `Function` [7], что делает функции довольно гибким инструментом при разработке приложений. Так, например, функцию можно присвоить переменной и вызывать её через эту переменную, а также передавать объявленную функцию в качестве входного аргумента в другую функцию.

Работа любого приложения, написанного на Dart, начинается с функции верхнего уровня `main`, которая представляет собой точку входа в приложение. В виду этого любой файл, в котором содержится эта функция, может выступать в роли того, с которого начинается запуск вашего приложения.

Говоря более общими словами, функции представляют собой инструмент, который позволяет вам переиспользовать код сколько угодно раз в процессе выполнения программы и разбивать сложные системы на составные части. Дополнительно к этому они являются альтернативой любимого метода начинающих программистов: «copy&paste».

3.1 Объявление функции

Для начала рассмотрим общий шаблон объявления функции:

```
[возвращаемый тип данных] имяФункции([ТипВходногоАргумента1  
                                         имяАргумента1, ..., n]){  
    // тело функции  
    [return возвращаемое значение]  
}
```

В квадратных скобках указаны необязательные элементы при объявлении функции. Если в таком языке, как C++ нам необходимо явно указать тип возвращаемого значения, то Dart может вывести его автоматически, в связи с чем его можно не указывать при объявлении функции. Несмотря на то, что это позволяет писать меньше кода, при большой кодовой базе такой подход может сыграть злую шутку, так как код проекта станем менее читаемым и понятным, особенно для новых разработчиков в проекте. В связи с этим лучше указывайте тип

возвращаемого значения, даже если функция ничего не возвращает (тогда пишем `void`).

Функции можно объявить практически в любой части кода. Обычно принято это делать на верхнем уровне модуля:

```
void main(List<String> arguments) {  
    myFunction(); // <- Привет!!!  
}  
  
void myFunction(){  
    print('Привет!!!');  
}
```

Когда при выполнении приложения в коде встречается вызов функции, то поток управления переходит в неё и выполняет её код, после управления возвращается в точку вызова функции и продолжается последовательная отработка кода основной программы.

Для передачи аргументов (параметров) в функцию необходимо после объявления её имени в круглых скобках указать тип и имя её входного аргумента, который свяжется с подаваемой при вызове функции на её вход переменной:

```
void myFunction(String name){  
    print('Привет, $name!');  
}  
  
void main(List<String> arguments) {  
    myFunction('Александр'); // <- Привет, Александр!  
}
```

Если функция должна возвращать значение, используйте оператор *return*:

```
String myFunction(String name){  
    var hello = 'Привет, $name!';  
    return hello;  
}  
  
void main(List<String> arguments) {  
    var myHelloString = myFunction('Александр');  
    print(myHelloString); // <- Привет, Александр!  
}
```

Для примера того, что компилятор Dart умеет выводить тип возвращаемого функцией значения, давайте модифицируем наш последний вариант кода следующим образом:

```
myFunction(String name){
  var hello = 'Привет, $name!';
  return hello;
}

void main(List<String> arguments) {
  var myHelloString = myFunction('Александр');
  print(myHelloString); // <- Привет, Александр!
}
```

Как видно из результата, удаление типа возвращаемого функцией значения не сказалось на работоспособности кода, но отразилось на его читаемости.

3.2 Объявление входных аргументов функции

Входные аргументы функции Dart подразделяются на:

- позиционные;
- именованные;
- необязательные позиционные;
- необязательные именованные;
- комбинация из вышеперечисленных.

Самыми простыми в понимании являются позиционные аргументы, так как последовательность передаваемых на вход функций переменных, должна соответствовать последовательности и типу объявленных в функции аргументов:

```
void myFunction(String name, int date, String monthName){
  print('$name родился $date $monthName!');
}

void main(List<String> arguments) {
  myFunction('Александр', 10, 'сентября');
}
// Александр родился 10 сентября!
```


Для объявления того, что на вход функции аргументы передаются именованным образом необходимо обернуть их в фигурные скобки «{}». После чего, в момент вызова функции необходимо явно указать имени какого аргумента какое значение передается. Если значение передаваемого аргумента не может хранить значение `null`, то необходимо перед объявляемым аргументом использовать ключевое слово `required`:

```
void myFunction({required String name,
                required int date,
                required String monthName}){
    print('$name родился $date $monthName!');
}

void main(List<String> arguments) {
    myFunction(date:10, name: 'Александр', monthName: 'сентября');
}
// Александр родился 10 сентября!
```

Обратите внимание на порядок передаваемых в функцию значений. Они могут передаваться в произвольной последовательности, так как мы явно указываем какому из аргументов функции будет соответствовать то или иное значение.

Если же значение передаваемого аргумента может хранить значение `null`, то после указания его типа необходимо добавить символ «?» и дополнительно предусмотреть проверку на `null`. `Null` может передаться в двух случаях: когда аргументу функции передали переменную, хранящую значение `null` или когда именованному аргументу ничего не передавали, вследствие чего значение аргумента по умолчанию становится равным `null`:

```
String myFunction({String? name,
                  required int date,
                  required String monthName}){
    if(name != null){
        return '$name родился $date $monthName!';
    }
    return 'Не установлено имя новорожденного!';
}

void main(List<String> arguments) {
    print(myFunction( date:10, monthName: 'сентября'));
```

```

}
// Не установлено имя новорожденного!

```

Когда мы указываем, что именованный аргумент может принимать значение `null`, он становится необязательным при вызове функции. Для того, чтобы указать необязательные аргументы при их позиционном размещении необходимо эти аргументы обернуть в квадратные скобки «[]»:

```

String myFunction(String name, int date, [String? monthName]){
    if(monthName != null){
        return '$name родился $date $monthName!';
    }
    return '$date числа, неустановленного месяца, родился $name!';
}

void main(List<String> arguments) {
    print(myFunction('Александр', 20));
    print(myFunction('Александр', 20, 'мая'));
}
// 20 числа неустановленного месяца родился Александр!
// Александр родился 20 мая!

```

Когда тип передаваемого в функцию аргумента не является примитивным типом данных, то есть – `List`, `Set`, `Map`, пользовательский тип данных и т.д., будьте предельно осторожны и не меняйте значения этих объектов в блоке кода функции, если эти изменения не требует логика работы приложения. Это связано с тем, что изменения останутся и после завершения функции. Говоря другими словами, примитивные типы данных передаются в функцию по значению, а все другие – по ссылке:

```

void add(List<int> funcList, int b){
    funcList.add(b);
}

void main(List<String> arguments) {
    var myList = [10, 20];
    add(myList, 3);
    print(myList); // [10, 20, 3]
}

```

3.3 Необязательные аргументы функции по умолчанию

По своей сути, когда мы объявляем необязательные позиционные или именованные аргументы они по умолчанию инициализируются значением `null`, если при вызове функции им не задается отличное от `null` значение. Если при их объявлении присвоим им значение, то оно будет использоваться в коде функции всякий раз, пока на вход функции явно не передастся другое значение:

```
String myFunction(String name, int date, [String monthName = 'июля']){
    return '$name родился $date $monthName!';
}

void main(List<String> arguments) {
    print(myFunction('Александр', 20));
    print(myFunction('Александр', 20, 'мая'));
}
// Александр родился 20 июля!
// Александр родился 20 мая!
```

При использовании позиционного расположения аргументов функции, те аргументы, которым присваиваются значения по умолчанию, должны указываться в конце. Следующие объявления функций, где значения по умолчанию задаются в других местах, приведут к ошибке:

```
String myFunction(String name = 'Александр', int date, String monthName){
    return '$name родился $date $monthName!';
}

String myFunction(String name, [int date=10, String monthName]){
    return '$name родился $date $monthName!';
}
```

В случае использования именованных аргументов для введения значения по умолчанию достаточно просто присвоить нужному вам аргументу, без ключевого слова `required`, это самое значение:

```
String myFunction({required String name,
    int date=10,
    required String monthName}){
    return '$name родился $date $monthName!';
}
```

```

void main(List<String> arguments) {
  print(myFunction(name:'Александр', monthName:'мая'));
  print(myFunction(date:14, name:'Александр', monthName:'мая'));
}
// Александр родился 10 мая!
// Александр родился 14 мая!

```

3.4 Область видимости переменных

В языке программирования Dart используется лексическая область видимости объектов. Это значит, что каждый блок кода имеет доступ к переменным, которые были объявлены на уровне выше, то есть «над» ним [8]. В качестве примера приведем вложенные функции:

```

var topLevel = 'Сверх-доступная переменная';
void topLevelFunction(){
  print(topLevel);
  var firstLevel= 'Не очень доступная переменная';
  void firstLevelFunction(){
    print(topLevel);
    print(firstLevel);
    var secondLevel= 'Так себе доступная переменная';
    void secondLevelFunction(){
      print(topLevel);
      print(firstLevel);
      print(secondLevel);
    }
    secondLevelFunction();
  }
  firstLevelFunction();
}

void main(List<String> arguments) {
  topLevelFunction();
}
// Сверх-доступная переменная
// Сверх-доступная переменная
// Не очень доступная переменная
// Сверх-доступная переменная
// Не очень доступная переменная
// Так себе доступная переменная

```

Если же попытаться из блока верхнего уровня обратиться к переменной, объявленной на уровне ниже, то это вызовет ошибку времени компиляции:

```
var topLevel = 'Сверх-доступная переменная';
void topLevelFunction(){
    print(topLevel);
    var firstLevel= 'Не очень доступная переменная';
    void firstLevelFunction(){
        print(topLevel);
        print(firstLevel);
        var secondLevel= 'Так себе доступная переменная';
    }
    print(secondLevel); // error: Undefined name 'secondLevel'.
}
void main(List<String> arguments) {
    topLevelFunction();
}
```

3.5 Обращение к функции через переменную

Как уже говорилось ранее — функцию можно присваивать (связывать с) переменной, после чего вызывать её посредством работы с этой самой переменной. Для примера напомним функцию сложения двух чисел, и будем с ней работать посредством переменной:

```
int add(int a, int b){
    return a + b;
}

void main(List<String> arguments) {
    var myAdd = add;
    print(myAdd(10,5)); // 15
}
```

3.6 Функция как входной аргумент другой функции

Теперь используем написанную функцию для сложения двух чисел в качестве входного аргумента другой функции, которая будет вычитать из передаваемого ей первого значения результат вычисления функции сложения. Так как нам необходимо указать тип аргумента, которым будет

выступать функция, для упрощения записи воспользуемся ключевым словом `typedef`:

```
typedef int MyFunctionAdd(int a, int b);

int add(int a, int b){
    return a + b;
}

int sub(int c, int a, int b, MyFunctionAdd func){
    return c - func(a, b);
}

void main(List<String> arguments) {
    print(sub(30, 21, 2, add)); // 7
}
```

Ключевое слово `typedef`, или как его еще называют – псевдоним типа функции, позволило нам в более компактной форме написать объявление функции и указать, что типу `MyFunctionAdd` соответствуют функции, которые принимают на вход два значения типа `int` и возвращают значение аналогичного типа данных. Без использования этого ключевого слова наш код выглядел бы следующим образом:

```
int add(int a, int b){
    return a + b;
}

int sub(int c, int a, int b, int Function(int a, int b) func){
    return c - func(a, b);
}

void main(List<String> arguments) {
    print(sub(30, 21, 2, add)); // 7
}
```

3.7 Анонимные и стрелочные функции

Помимо функций, которые содержат их явные названия, вы можете создавать и анонимные функции (порой их еще называют «Лямбда-функции»), то есть функции без имени. Их структуру записи можно представить следующим образом:

```

([Type] arg1[, ...]) {
    // блок кода
};

```

Эти функции на свой вход могут принимать сколько угодно значений или не принимать их вовсе. Всё зависит от того, каким образом она была объявлена. При этом тип входного аргумента в некоторых случаях можно опускать. Для примера давайте создадим анонимную функцию для вывода элементов списка и передадим её в метод списка `forEach`:

```

void main(List<String> arguments) {
    var myList = ['Привет!', 'Я', '-', 'анонимная', 'функция!'];
    myList.forEach((item) {
        print('По индексу ${myList.indexOf(item)} хранится значение => $item');
    });
}
// По индексу 0 хранится значение => Привет!
// По индексу 1 хранится значение => Я
// По индексу 2 хранится значение => -
// По индексу 3 хранится значение => анонимная
// По индексу 4 хранится значение => функция!

```

Анонимные функции поддерживают механизм зацепления. Например, внутри тела функции, из предыдущего примера, нам доступны переменные, объявленные в основном блоке кода функции `main`, до момента объявления самой анонимной функции. Более подробно механизм зацепления мы рассмотрим в следующем разделе.

Стрелочная функция предоставляет форму для объявления однострочных именованных или анонимных функций. Их поведение аналогично обычным функциям за тем исключением, что они по умолчанию всегда возвращают значение, то есть оператор `return` в этих функциях не используется, но его наличие подразумевается. Давайте для примера перепишем один из ранее рассмотренных примеров:

```

typedef int MyFunctionAdd(int a, int b);

int add(int a, int b) => a + b;

int sub(int c, int a, int b, MyFunctionAdd func) => c - func(a, b);

void main(List<String> arguments) {

```

```
    print(sub(30, 21, 2, add)); // 7
}
```

Анонимные или стрелочные функции могут присваиваться переменной, точно также как и обычные функции:

```
int add(int a, int b) => a + b;
```

```
void main(List<String> arguments) {
    var newAddFunction = add;
    var newSubFunction = (int c, int a, int b, MyFunctionAdd func) {
        return c - func(a, b);
    };
    print(newSubFunction(30, 21, 2, newAddFunction)); // 7
}
```

3.8 Замыкания

Замыкания представляют собой довольно мощный инструмент, в основе которого лежит возможность функций запоминать значения переменных из объемлющих областей видимости. То есть из тех областей видимости, где данная функция была объявлена. В основе идеи замыкания лежит то, что функция может возвращать функцию, которая в свою очередь может на вход принимать совершенно отличные значения от тех, что подаются функции верхнего уровня, но использует в своей работе данные, определенные в функции верхнего уровня. Звучит немного запутанно, не прав да ли?

Давайте представим, что в качестве функции верхнего уровня выступает завод, производящий технику. Ему каждый день поставляют оборудование и забирают изготовленные микроволновые печи, которые развозят по магазинам и, в конечном счете, одна из них оказалась в доме покупателя. Каждый из нас часто пользуется этим устройством, чтобы разогреть или приготовить еду и даже не задумывается, а из каких деталей она состоит, как они взаимодействуют друг с другом и т. д. Так вот, эта микроволновая печь и есть возвращаемая вложенная функция. В неё мы ставим тарелку с супом, задаем ей режим работы, а она уже выполняет нагрев с использованием той аппаратной начинке, что использовалось на производственной линии завода:


```
int index_microwave = 0;
```

```
Function factory(String name_microwave, int power){  
    var model = '$name_microwave-RX-0003$index_microwave';  
    index_microwave++;  
    return (String dish, int mode){  
        var myStr = StringBuffer('Микроволновка "$model" мощностью $power Вт');  
        myStr.write(', греет блюдо "$dish" в режиме $mode');  
        return myStr;  
    };  
}
```

```
void main(List<String> arguments) {  
    var microwave = factory('Scarlet', 750);  
    print(microwave('Борщ', 3));  
    print(microwave('Котлеты', 5));  
    var newMicrowave = factory('Scarlet', 1000);  
    print(newMicrowave('Пагу', 2));  
}  
// Микроволновка "Scarlet-RX-00030" мощностью 750 Вт, греет  
// блюдо "Борщ" в режиме 3  
// Микроволновка "Scarlet-RX-00030" мощностью 750 Вт, греет  
// блюдо "Котлеты" в режиме 5  
// Микроволновка "Scarlet-RX-00031" мощностью 1000 Вт, греет  
// блюдо "Пагу" в режиме 2
```

Также замыкания могут применяться в программах, которым необходимо генерировать обработчики событий на лету в ответ на условия, сложившиеся во время выполнения.

В качестве еще одного примера замыкания реализуем функцию, которой на вход подается степень, в которую будем возводить числа, подаваемые на вход возвращаемой ей функции:

```
import 'dart:math'; // подключаем библиотеку math  
// для использования функции pow  
typedef int MyPow(int value);
```

```
MyPow Degree(int degree){  
    return (int value) => pow(value, degree).toInt();  
}
```

```
void main(List<String> arguments) {
```

```

var calculation = Degree(3);
print(calculation(3)); // 27
print(calculation(2)); // 8
calculation = Degree(8);
print(calculation(3)); // 6561
print(calculation(7)); // 5764801
}

```

3.9 Рекурсия

В теле функции может вызываться эта же самая функция. Такой механизм называется рекурсией. Он обычно используется как альтернатива цикла, когда нам необходимо обходить структуры, которые имеют произвольную непредсказуемую форму и глубину.

Существует несколько видов организации рекурсии: прямая и косвенная. Давайте разберемся с тем, как они организуются на примере задачи суммирования элементов последовательности:

```

int addFunction(List<int> myList){
    print(myList);
    if (myList.length <=1){
        return myList[0];
    }
    else{ return myList[0] + addFunction(myList.sublist(1)); }
}

```

```

void main(List<String> arguments) {
    var myList = [10, 20, 30, 5, 3, 2];
    print(addFunction(myList));
}
// [10, 20, 30, 5, 3, 2]
// [20, 30, 5, 3, 2]
// [30, 5, 3, 2]
// [5, 3, 2]
// [3, 2]
// [2]
// 70

```

В примере выше у нас приведена прямая рекурсия, когда функция явно вызывает саму себя. Каждый раз при вызове функцией самой себя ей на вход подается срез списка, содержащий только те элементы, которые еще не участвовали в операции суммирования. Теперь

реализуем суммирование элементов списка посредством косвенной рекурсии:

```
int addFunction(List<int> myList){
    print(myList);
    if (myList.length <=1){
        return myList[0];
    }
    else{
        return anotherFunction(myList);
    }
}

int anotherFunction(List<int> myList){
    return myList[0] + addFunction(myList.sublist(1));
}

void main(List<String> arguments) {
    var myList = [10, 20, 30, 5, 3, 2];
    print(addFunction(myList));
}
// [10, 20, 30, 5, 3, 2]
// [20, 30, 5, 3, 2]
// [30, 5, 3, 2]
// [5, 3, 2]
// [3, 2]
// [2]
// 70
```

Основным требованием при организации рекурсии является наличие условия выхода из неё. В случае его отсутствия произойдет переполнение стека вызова функций (превышение лимита по количеству рекурсивного вызова функции) и выполнение кода прервется, оповестив пользователя о наличии проблемы следующим исключением: «Unhandled exception: Stack Overflow».

3.10 Генераторные функции

Генераторные функции используются для ленивой генерации последовательности значений по запросу. Они выступают отличным подспорьем спискам, так мы в этом случае не храним в памяти массив значений, а создаем объект с необходимым нам текущим значением

генерируемой последовательности только в момент обращения к генераторному объекту.

Dart предоставляет два варианта реализации генераторных функций:

- Синхронная генераторная функция – функция, возвращающая объект типа `Iterable<T>`, где `T` – тип генерируемого функцией значения. Класс `Iterable` представляет собой интерфейс, реализуемый итерируемым набором значений или «элементов», к которым можно получить последовательный доступ. Для обозначения синхронной генераторной функции, после её объявления, перед телом самой функции она помечается как `<sync*>`.
- Асинхронная генераторная функция – функция, возвращающая объект типа `Stream<T>`, где `T` – тип генерируемого функцией значения. Класс `Stream` представляет собой источник событий асинхронных данных. Для обозначения асинхронной генераторной функции, после её объявления, перед телом самой функции она помечается как `<async*>`. Более подробно асинхронное программирование обсудим в одном из следующих разделов.

Еще одной отличительной чертой обычных функций от генераторных является то, что для возвращения из неё значения используется оператор `yield`, а не `return`. Именно благодаря этому оператору генераторные функции автоматически приостанавливают и возобновляют свое выполнение и состояние вокруг точки генерации значений, а не прекращают свою работу, как обычные функции. В момент приостановки выполнения генераторной функции сохраняется информация о её состоянии, куда входят данные о местоположении точки выхода из функции и локальной области видимости. Возобновляется работа генераторной функции с оператора `yield`, то есть с той точки, в которой была выполнена остановка её выполнения.

В качестве примера давайте сгенерируем последовательность значений от 0 до 5 и запишем её в список:

```
Iterable<int> myGenerator() sync* {  
  var k = 0;  
  while (k < 5) {  
    yield k++;  
  }  
}
```

```

    }
}

void main(List<String> arguments) {
    var result = <int>[];
    for(var it in myGenerator()){
        result.add(it);
    }
    print(result); // [0, 1, 2, 3, 4]
}

```

Для того, чтобы лучше понять принцип работы генераторной функции перепишем её код следующим образом:

```

Iterable<int> myGenerator() sync* {
    yield 0;
    yield 1;
    yield 2;
    yield 3;
    yield 4;
}

void main(List<String> arguments) {
    var result = <int>[];
    for(var it in myGenerator()){
        result.add(it);
    }
    print(result); // [0, 1, 2, 3, 4]
}

```

Как видно из примера, при первом проходе цикла `for` из генераторной функции вернулось значение 0, после чего её работа приостановилась до момента второго прохода цикла. При последующем обращении к функции она продолжила свою работу с точки предыдущего выхода из неё, то есть с «`yield 0;`». После того, как последовательность операторов `yield` в теле функции закончилась, прекратился и цикл `for`.

Теперь давайте реализуем генераторную функцию, которая будет возвращать только значения, которые нацело делятся на 4, а диапазон по какое значение должна генерироваться последовательность будет задаваться пользователем. В этом нам поможет библиотека «`dart:io`»:

```

import 'dart:io';

Iterable<int> myGenerator(int n) sync* {
  var k = 0;
  while(k < n){
    if (k % 4 == 0){
      yield k;
    }
    k++;
  }
}

void main(List<String> arguments) {
  var result = <int>[];

  print('Введите правую границу генерируемой последовательности: ');
  // читаем значение введенное с клавиатуры
  var n = int.parse(stdin.readLineSync()); // вводим 20

  for(var it in myGenerator(n)){
    result.add(it);
  }
  print(result); // [0, 4, 8, 12, 16]
}

```

При запуске этого кода в VS Code может возникнуть проблема, которая связана в невозможности плагина переопределить поток ввода. Для того, чтобы запустить код необходимо будет создать в директории «имя_проекта\.vscode» файл «launch.json» и добавить в него следующую конфигурацию:

```

"configurations": [
  {
    "name": "имя_запускаемого_файла",
    "request": "launch",
    "type": "dart",
    "console": "terminal"
  }
]

```

После чего необходимо перезапустить VS Code. Всё, теперь можно запустить код с использованием горячих клавиш «Ctrl+F5» (Запуск без использования режима отладки) и ввести с клавиатуры желаемое числовое значение.

Далее рассмотрим, как можно использовать генераторную функцию вне циклов, осуществляя работу через возвращаемый ей при создании объект типа `Iterable<T>`:

```
Iterable<int> myGenerator(int n) sync* {
    var k = 0;
    while(k < n){
        if (k % 4 == 0){
            yield k;
        }
        k++;
    }
}

void main(List<String> arguments) {
    var result = <int>[];
    var it = myGenerator(20);
    it.forEach((element) {result.add(element);});
    var result1 = it.toList();
    print(result); // [0, 4, 8, 12, 16]
    print(result1); // [0, 4, 8, 12, 16]
}
```

Для более подробной информации о том, какие возможности предоставляет класс `Iterable` обратитесь к официальной документации.

Теперь рассмотрим способ создания асинхронной генераторной функции, выводящей в терминал значения генерируемой последовательности:

```
Stream<int> myAsyncGenerator(int n) async* {
    var k = 0;
    while(k < n){
        if (k % 4 == 0){
            yield k;
        }
        k++;
    }
}
```

```
void main(List<String> arguments) {
    Stream<int> sequence = myAsyncGenerator(30);
    sequence.listen(print);    // 0 4 8
}
```

Пользоваться асинхронными генераторными функциями без знания принципов асинхронного программирования не стоит. На данный момент пока ограничьтесь использованием их синхронных вариантов.

Резюме по разделу

В данном разделе мы рассмотрели, как объявлять и использовать функции при написании кода. Какие способы передачи аргументов в функции существуют и чем отличаются. Что такое замыкания и как они реализованы в Dart.

Отдельно стоит отметить такой механизм, как генераторные функции, которые используются для ленивой генерации последовательности значений по запросу. Они представляют собой альтернативу спискам, так как при их использовании в памяти не хранится массив значений и объект, с необходимым нам текущим значением элемента последовательности, создается только в момент обращения к генераторному объекту.

Особенно осторожно стоит обращаться с такими типами передаваемых аргументов в функцию, как: `List`, `Set`, `Map`, пользовательский тип данных и т.д. Это связано с тем, что такие аргументы передаются в функцию по ссылке, а не значению и какое-либо их изменение в теле функции приведет к тому, что они останутся и после завершения функции. Поэтому будьте предельно осторожны и не меняйте значения этих объектов в блоке кода функции, если эти изменения не требует логика работы приложения.

Вопросы для самопроверки

1. Для чего используются функции?
2. Назовите функцию верхнего уровня, которая представляет собой точку входа в приложение?
3. Перечислите все способы объявления аргументов функции.

4. Чем отличаются позиционные от именованных аргументов функции?
5. Для чего используется ключевое слово **required**?
6. Какая область видимости используется в Dart? Приведите пример.
7. Как обратиться к функции через переменную?
8. Можно ли функцию использовать в качестве входного аргумента другой функции? Приведите пример.
9. Для чего используется ключевое слово **typedef**?
10. Что такое анонимные и стрелочные функции? Для чего они используются?
11. Может ли функция в качестве возвращаемого значения возвращать другую функцию? Приведите пример.
12. Что представляет собой механизм замыкания и зачем им пользоваться?
13. Перечислите виды организации рекурсии. Чем они отличаются?
14. Для чего используются генераторные функции?
15. Какие варианты реализации генераторных функций существуют в Dart?
16. Чем оператор **yield** отличается **return**?
17. Что лучше использовать: списки или генераторные функции? В каких случаях?

Упражнения

Напишите функцию:

1. для вычисления максимального из трех чисел;
2. возвращающую сумму элементов списка;
3. возвращающую произведение элементов списка;
4. удаляющую повторяющиеся элементы в списке;
5. вычисляющую факториал задаваемого числа;
6. проверяющую вхождение элемента в список или множество;
7. выводящую в терминал элементы списка с нечетным индексом;
8. инвертирующую последовательность расположения элементов в списке;
9. вычисляющую среднеарифметическое значение элементов списка;
10. подсчитывающую количество вхождения в список, передаваемого на вход функции значения.

11. подсчитывающую произведение элементов списка с использованием механизма рекурсии.

Напишите генераторную функцию:

1. выводящую в терминал значения от 10 до 33;
2. выводящую в терминал значения от 1 до 35 с шагом 7;
3. возводящую в квадрат генерируемые значения из диапазона от 5 до 12;
4. выводящую в терминал последовательность вещественных значений от 0 до 1, с задаваемым шагом. Например: 0.1.

4 Библиотеки и пакеты

При написании многократно используемого кода его принято выделять в отдельные модули, представляющие собой единицу организации программ наивысшего уровня, которая упаковывает программный код, данные и предоставляет изолированные пространства имен, цель которых - свести к минимуму конфликты имен переменных внутри программ.

Для модульного представления кода вашего проекта в Dart используются библиотеки и пакеты. Библиотека может представлять собой как один файл с кодом, подключаемый к основному приложению и расположенный в той же директории, так и набор таких файлов, организованных в каталог, с единой или множественной точкой доступа к ним, позволяющей скрыть детали реализации. Любое приложение Dart также является библиотекой.

Пакет представляет собой каталог, который может содержать в себе любое количество библиотек. Обязательным условием при создании пакета является наличие в нем файла `pubspec.yaml` на верхнем уровне пакета, содержащем важную информацию о самом пакете и его зависимостях от других пакетов.

Dart предоставляет некоторый базовый набор библиотек, который можно подключить посредством оператора `import`:

- `dart:core`. Данная библиотека подключается автоматически в каждый dart-файл и предоставляет доступ к встроенным типам, коллекциям и другим основным функциям.
- `dart:async`. Эта библиотека используется для поддержки асинхронного программирования.
- `dart:math`. Предоставляет доступ к математическим константам, функциям и генератору случайных чисел.
- `dart:convert`. Используется для преобразования между различными представлениями данных.
- `dart:html`. Предоставляет доступ к DOM (Document Object Model) html-файлов и другим API для браузерных приложений.
- `dart:io`. Служит для ввода и вывода данных. Позволяет работать с файлами, каталогами, сокетами, процессами, писать собственные серверные и клиентские части приложений и т. д.

4.1 Импортирование кода из файла с расширением «.dart»

Для начала разберемся с тем как подключать (импортировать) код из другого файла с расширением «.dart», расположенного в основной директории разрабатываемого приложения. В качестве примера реализуем файл, который содержит функции операций над числами: умножение, деление, сложение и т. д.

На первом шаге создадим простой консольный проект («Simple Console Application»), имеющий следующую структуру и имя:

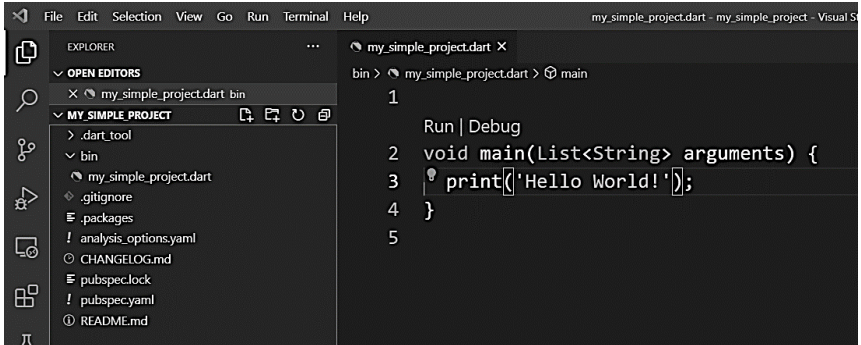


Рисунок 4.1 – Структура созданного приложения

Далее в директории «bin», где расположен файл «my_simple_project.dart» создадим директорию «src», куда добавим новый файл с названием «my_calculator.dart»:

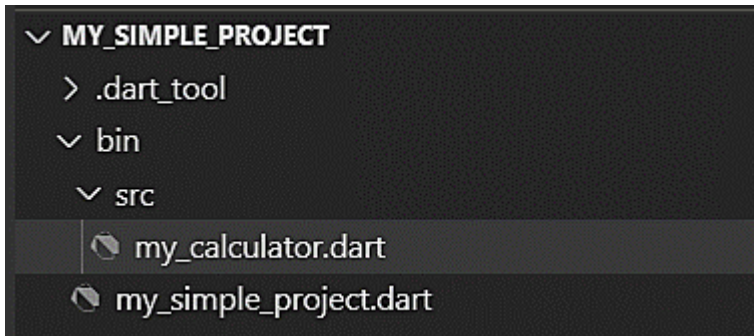


Рисунок 4.2 – Добавление файла «my_calculator.dart» в проект

Теперь можно приступить к его заполнению кодом:
`import 'dart:math';`

```
double add(double a, double b) => a + b;
double sub(double a, double b) => a - b;
double div(double a, double b) => a / b;
double mul(double a, double b) => a * b;
double pow2(double a) => a * a;
double powN(double a, double n) => pow(a, n).toDouble();
```

Импортируем код из файла «my_calculator.dart» в файл «my_simple_project.dart» и выведем в терминал результаты выполнения некоторых из функций:

```
import 'src/my_calculator.dart';

void main(List<String> arguments) {
  print(add(3.5, 10)); // 13.5
  print(mul(2.5, 4)); // 10.0
  print(pow2(3)); // 9.0
}
```

Как видим из результата выполнения кода, функции, объявленные в файле «my_calculator.dart» доступны в основном файле приложения по их имени. Такое поведение при импортировании не всегда бывает удобным. Особенно, когда несколько импортируемых файлов будут содержать одинаковое объявление какой-либо функции или другого объекта — это приведет к ошибке. Для того, чтобы в этом убедиться создадим файл «short_calculator.dart» в который скопируем функцию сложения и вычитания, после чего импортируем его:

```
import 'src/my_calculator.dart';
import 'src/short_calculator.dart';

void main(List<String> arguments) {
  print(add(3.5, 10)); // error: The name 'add' is defined in the libraries
  print(mul(2.5, 4));
  print(pow2(3));
}
```

Чтобы избежать таких ошибок после объявления импорта одной из библиотек можно использовать ключевое слово **as** (префикс/Prefix) за которым следует указать имя, посредством которого мы будем обращаться к функциям из библиотеки:

```
import 'src/my_calculator.dart' as calculator;
import 'src/short_calculator.dart';

void main(List<String> arguments) {
  print(add(3.5, 10)); // вызов функции из short_calculator.dart
  print(calculator.mul(2.5, 4));
  print(calculator.pow2(3));
  print(calculator.add(2.5, 4)); // вызов функции из my_calculator.dart
}
```

Часть функций в библиотеке можно объявлять приватными путем добавления символа нижнего подчеркивания перед именем функции. Их можно использовать в самой библиотеке, но нельзя импортировать. Для примера перепишем файл «my_calculator.dart» следующим образом:

```
import 'dart:math';

double add(double a, double b) => _add(a, b);
double sub(double a, double b) => a - b;
double div(double a, double b) => a / b;
double mul(double a, double b) => a * b;
double pow2(double a) => a * a;
double powN(double a, double n) => pow(a, n).toDouble();

double _add(double a, double b){
  return (a + b) * 10;
}
```

И попробуем средствами IDE обратиться к приватной функции библиотеки:

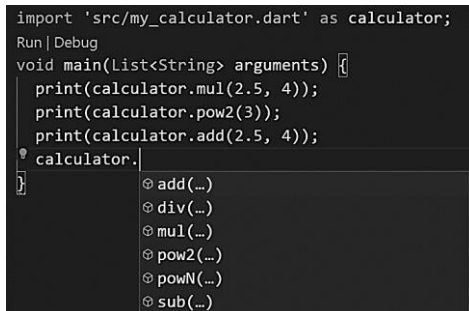


Рисунок 4.3 – Доступные для использования функции библиотеки

Как видим из рисунка 4.3 доступ к приватным функциям импортируемых библиотек из основного кода приложения закрыт. Попытка вызова таких функций приведет к ошибке:

```
import 'src/my_calculator.dart' as calculator;

void main(List<String> arguments) {
  print(calculator.mul(2.5, 4));
  print(calculator.pow2(3));
  print(calculator.add(2.5, 4));
  calculator._add(2.5, 4); // error: The function '_add' isn't defined.
}
```

4.2 Импортирование части функциональности

Порой бывают ситуации, когда из имеющейся в нашем распоряжении библиотеки необходимо использовать одну или две функции, либо же закрыть к одной из имеющихся функций доступ, чтобы не было возможности к ней обратиться из кода, где подключили библиотеку. На эти случаи Dart предоставляет следующие ключевые слова:

- **show** – позволяет указать используемую часть импортируемой библиотеки. К остальным частям доступ будет закрыт;
- **hide** – скрывает указанную часть импортируемой библиотеки, к которой будет закрыт доступ. Остальные части библиотеки будут доступны для использования.

Данные ключевые слова можно использовать как с ключевым словом **as** (после его объявления), так и без него.

4.2.1 Использование ключевого слова **show**

Для демонстрации принципа работы частичного импортирования с использованием ключевого слова **show** возьмем предыдущий пример кода и укажем, что из импортируемой библиотеки нам нужна только функция умножения:

```
import 'src/my_calculator.dart' as calculator show mul;
Run | Debug
void main(List<String> arguments) {
  print(calculator.mul(2.5, 4));
  calculator.
}
mul(...) (double
```

Рисунок 4.4 – Импортрование части библиотеки

Если необходимо импортировать несколько частей библиотеки, то их необходимо перечислить через запятую:

```
import 'src/my_calculator.dart' as calculator show mul, add;
```

```
void main(List<String> arguments) {
  print(calculator.mul(2.5, 4)); // 10.0
  print(calculator.add(2.5, 4)); // 65.0
}
```

Если результат от вызова функции сложения вам кажется подозрительным, то вспомните, что сейчас она вызывает выполнение приватной функции `_add`:

```
double _add(double a, double b){
  return (a + b) * 10;
}
```

4.2.2 Использование ключевого слова `hide`

Для демонстрации принципа работы частичного импортирования с использованием ключевого слова `hide` укажем, что запрещаем импортировать функцию сложения и умножения:

```
import 'src/my_calculator.dart' as calculator hide mul, add;
Run | Debug
void main(List<String> arguments) {
  calculator.
}
div(...) (double a, double b)
pow2(...)
powN(...)
sub(...)
class
```

Рисунок 4.5 – Импортрование части библиотеки

Как видно из рисунка 4.5 после такого импортирования мы не можем использовать указанные при импортировании функции: `mul` и `add`.

4.3 Отложенная загрузка импортируемого файла или библиотеки

Отложенная (ленивая) загрузка библиотеки позволяет приложению загружать библиотеку по запросу. То есть, такая библиотека может в процессе работы приложения и не загружаться вовсе, если используются операторы управления потоком выполнения. Для объявления отложенной загрузки импортируемой библиотеки используется ключевое слово `deferred` в связке с префиксом `as <name>`.

Дополнительным условием использования данного механизма является оборачивание отложенной загрузки библиотеки в асинхронную функцию, возвращающую `Future`. Это связано с тем, что используемый метод для самой ленивой загрузки – `loadLibrary` возвращает этот тип данных.

Для примера в качестве примера выполним отложенную загрузку библиотеки и вызовем функцию деления:

```
import 'src/my_calculator.dart' deferred as calculator;
```

```
void main(List<String> arguments) {  
  callLibrary(34, 5);  
}
```

```
Future callLibrary(double a, double b) async{  
  await calculator.loadLibrary();  
  print(calculator.div(a, b)); // 6.8  
}
```

При объявлении отложенной загрузки можно использовать ключевые слова `show` и `hide` для регулирования доступа к частям импортируемой библиотеки.

4.4 Создание и использование библиотеки

Dart существует негласное соглашение относительно библиотек. Несмотря на то, что создаваемая библиотека может иметь любую иерархию каталогов с кодом, рекомендуется весь код, содержащий

реализацию, помещать в каталог «lib/src». Данный каталог является закрытым и не рекомендуется явным образом импортировать из него файлы с кодом. Для того, чтобы пользователь имел возможность взаимодействовать с реализацией библиотеки, необходимо в самой директории «lib» поместить файлы, которые экспортируют файлы с кодом из «lib/src» и представляют собой API вашей библиотеки.

Для начала создадим новый простой консольный проект с названием «library_test», после чего добавим каталог «lib», внутри которого еще один каталог «src». Следующим действием добавим внутрь каталога «lib/src» файлы с кодом функций:

```
//my_add.dart
double add(double a, double b) => a + b;

//sub.dart
double sub(double a, double b) => a - b;

//mul.dart
double mul(double a, double b) => a * b;

//my_pow_n.dart
import 'dart:math';

double powN(double a, double n) => pow(a, n).toDouble();
```

При импорте таких файлов из библиотеки в клиентском коде приложения необходимо в начале указания пути использовать директиву «package:». Если мы в одном файле библиотеки импортируем другой, который расположен так же в библиотеке, то необходимо использовать относительные пути (например, как в предыдущих примерах).

Любой из объявленных файлов мы можем импортировать из библиотеки следующим образом:

```
import 'package:library_test/src/my_add.dart';
void main(List<String> arguments) {
  print(add(10, 34.2)); // 44.2
}
```

В данном случае после директивы **package:** указывается название текущего приложения. Таким образом остальная часть пути после

«package:library_test» берется из каталога «lib». Но приведенный способ импортирования файлов из библиотеки не является рекомендуемым, так как у нас нет файла, описывающего API библиотеки, и мы получаем прямой доступ к реализациям файлов библиотеки.

Для того, что бы привести структуру библиотеки в соответствии с рекомендациями [10] в каталоге «lib» создадим файл «calculator.dart», в котором укажем экспортируемые из библиотеки файлы:

```
export 'src/my_add.dart';  
export 'src/my_mul.dart';  
export 'src/my_pow_n.dart';  
export 'src/my_sub.dart';
```

Теперь можем переписать предыдущий код следующим образом:

```
import 'package:library_test/calculator.dart';
```

```
void main(List<String> arguments) {  
  print(add(10, 34.2)); // 44.2  
  print(mul(10, 4.2)); // 42.0  
  print(sub(10, 4.8)); // 5.2  
  print(powN(10, 3)); // 1000.0  
}
```

Как видно из кода, мы одной строчкой импортировали все доступные функции, которые объявляли в различных файлах библиотеки и прописали, как экспортируемые в файле «calculator.dart» каталога «lib».

При таком импортировании библиотеки доступны все ранее рассмотренные механизмы, когда используются ключевые слова: **show**, **hide**, **as** и **deferred as**.

В каталоге «lib» можно создавать неограниченное количество файлов, которые могут предоставлять доступ либо к части, либо ко всем файлам библиотеки. Для примера создадим еще один файл, который назовем «short_calculator.dart» и посредством него экспортируем файлы библиотеки только для сложения и вычитания:

```
export 'src/my_add.dart';  
export 'src/my_sub.dart';
```

В случае импортирования этого файла нам станут доступны всего лишь 2 функции, находящиеся в нашей библиотеке:

```
import 'package:library_test/short_calculator.dart' as calc;

Run | Debug
void main(List<String> arguments) {
  print(calc.add(10, 34.2)); // 44.2
  print(calc.sub(10, 4.8)); // 5.2
  calc.
}
  add(...)    (double a, double b) ...
  sub(...)
```

Рисунок 4.6 – Подключение библиотеки в клиентском коде приложения

Текущий проект «library_test» можно рассматривать как уже готовый пакет, который при необходимости можем подключить к другому проекту. Это связано с тем, что на его верхнем уровне имеется файл «pubspec.yaml», с информацией о самом пакете и его зависимостях от других пакетов.

4.5 Подключение пакета к проекту

Создадим новый простой консольный проект с именем «packages_test» и в его файле «pubspec.yaml» установим зависимость в качестве написанного ранее проекта «library_test» с указанием относительного пути его расположения (то есть от того места, где создается текущий проект):

```
name: packages_test
description: A simple command-line application.
# version: 1.0.0
# homepage: https://www.example.com
environment:
  sdk: '>=2.12.0 <3.0.0'
```

Устанавливаем зависимости проекта

dependencies:

На этом уровне имена подключаемых пакетов

library_test: **# Имя подключаемого пакета**

Настройки пакета

path: ../library_test **# Путь до пакета**

```
dev_dependencies:  
  pedantic: ^1.10.0
```

В моем случае путь до пакета «`library_test`» выглядит следующим образом «`../library_test`». Это связано с тем, что проект «`packages_test`» был создан в той же директории, что и «`library_test`».

Теперь импортируем из подключенного пакета файл «`calculator.dart`», который наряду с файлом «`short_calculator.dart`» предоставляет API работы с подключаемым пакетом:

```
import 'package:library_test/calculator.dart' as calc;  
  
void main(List<String> arguments) {  
  print(calc.add(10, 34.2)); // 44.2  
  print(calc.mul(10, 4.2)); // 42.0  
  print(calc.sub(10, 4.8)); // 5.2  
  print(calc.powN(10, 3)); // 1000.0  
}
```

Если имеется необходимость установить пакет, который находится в менеджере пакетов `pub`, то достаточно в зависимостях (`dependencies:`) файла «`pubspec.yaml`» прописать имя этого пакета и его версию. Также имеется возможность производить установку пакетов, входящих в зависимость, напрямую из `github`. Для более подробной информации по вариантам подключения пакетов обратитесь к официальной документации [11].

Резюме по разделу

В данном разделе мы рассмотрели как можно импортировать функции, переменные и т.д., написанные в других модулях (файлах с расширением «`.dart`»), писать свои библиотеки и пакеты. Каждое приложение, написанное на Dart, может рассматриваться как уже готовая к использованию библиотека (пакет), импортируемая в другой проект довольно простым способом.

При написании собственной библиотеки помещайте её код в каталог «`lib/src`», а в каталоге «`lib`» организуйте файлы, экспортирующие

необходимую функциональность для использующей стороны этой библиотеки, тем самым организовав API.

Вопросы для самопроверки

1. Какой набор базовых библиотек входит в Dart?
2. Какая базовая библиотека автоматически подключается к dart-файлам?
3. Какое ключевое слово позволяет подключать к вашему модулю код из других модулей (файлов с расширением «.dart»)?
4. Для чего используется файл «pubspec.yaml»?
5. Как создать в проекте библиотеку?
6. Как организовать API для доступа к предоставляемой библиотекой функциональности?
7. Для чего используются ключевые слова `show` и `hide`?
8. Как организовать отложенную загрузку подключаемой библиотеки/модуля?
9. Как подключить пакет к проекту?

Упражнения

1. Напишите библиотеку предоставляющую следующую функциональность: вычисление максимального из трех чисел, расчет суммы и произведения элементов списка.
2. Напишите библиотеку предоставляющую следующую функциональность: удаление повторяющихся элементы в списке, расчет факториал задаваемого числа, проверка наличия элемента в списке или множестве.
3. Напишите библиотеку предоставляющую следующую функциональность: подсчет количества элементов в списке, которые нацело делятся на 4 и 3, расчет среднеарифметического значения элементов списка.
4. Напишите библиотеку предоставляющую следующую функциональность: создание списка, состоящего из последовательности целых (от 23 до 47) и вещественных значений от 0 до 1, с задаваемым шагом.
5. Напишите библиотеку предоставляющую следующую функциональность: подсчет количества вхождения каждого элемента в список, инверсия последовательности расположения элементов в списке.

5 Объектно-ориентированное программирование в Dart

В основе объектно-ориентированного программирования (ООП) лежит понятие – объект, который представляет собой некоторую сущность предметной области, в рамках которой ведется разработка приложения. В качестве примера давайте возьмем ветеринарную клинику. Здесь объектами могут выступать сотрудники, пациенты (коты, собаки и т.д.), хозяева животных, оборудование ветклиники и т.д.

Могут ли являться объектами одушевленные предметы (люди, животные)? Да, могут. Это связано с тем, что они являются частью предметной области. В нашем случае – ветеринарной клиники.

Итак, на уровне предметной области объекты являются её составными частями, а на уровне программных систем объекты описываются классами. При том не всегда один класс описывает только один объект. Всё зависит от структуры того объекта, который необходимо перенести в программный код, в связи с чем некоторые объекты могут быть описаны не одним, а целым набором классов.

Каждый объект в ООП характеризуется состоянием и поведением. Под состоянием понимается некоторый набор атрибутов, которые позволяют нам описать из чего состоит объект и какие характеристики ему присущи. В качестве примера могут выступать: имя, возраст, пол, адрес у сотрудника ветклиники или кличка, возраст, пол, флаг состояния стерилизации животного (`true` либо `false`) у условного кота.

Говоря простыми словами – **состояние объекта**, при его описании классом на языке программирования, переводится в **переменные реализуемого класса**.

Под поведением понимается некоторый набор действий, которые объект может совершать над своим внутренним состоянием. Так, например, кот может: бегать, кушать, играть, спать и т. д. То есть **поведение объекта**, при его описании классом, переводится в **методы реализуемого класса**.

В основе ООП лежат 3 принципа:

- Инкапсуляция;
- Наследование;
- Полиморфизм.

Под инкапсуляцией понимается возможность объекта скрыть часть своего состояния или поведения (объявить, как приватные) от других объектов за интерфейсом класса (публичные методы). То есть с объектом можно будет работать только через его публичные методы (интерфейс класса). Инкапсуляция распространяется не только на классы, но и на всё программируемое приложение в целом. Так, например, вы не должны предоставлять возможность напрямую обращаться в ряды состояний модуля, библиотеки и т.д., а предоставлять интерфейс для работы с ними. Это связано с тем, что при наличии возможности прямого изменения состояния (не через методы) могут возникать трудно отлавливаемые ошибки и появляются проблемы в архитектуре, которые в последствии не позволят достаточно быстро адаптировать приложение под новые требования заказчика по его функциональности.

Наследование позволяет создавать новые классы на основе существующих. Тот класс, от которого происходит наследование принято называть базовым классом (суперклассом, родительским классом), а класс, который наследуется от базового – производным классом (подклассом, дочерним классом). Производный класс наследует состояние и поведение базового класса, что позволяет повторно использовать имеющийся код, но при этом необходимо всегда придерживаться интерфейса, что задает базовый класс. Также производный класс может объявлять собственные переменные и методы, тем самым расширяя состояние и поведение базового класса.

Полиморфизм очень тесно связан с наследованием и позволяет менять (переопределять) поведение базовых классов в производных. То есть один метод в зависимости от класса: базовый или производный, может иметь несколько реализаций. В качестве примера рассмотрим ситуацию, когда у нас имеется класс **Животное**, где задается метод **приветствоватьХозяина**. Этот метод может как иметь реализацию, так и не иметь. В случае, если объявленный метод в классе не имеет реализации он считается чисто виртуальным (абстрактным), а сам класс будет называться абстрактным классом, чьи экземпляры нельзя явным образом создать в системе (рассмотрим это позже). У базового класса **Животное** имеется 2 производных класса: **Кот** и **Собака**, где у каждого своя реализация метода **приветствоватьХозяина** («Мяу» и «Гав»). И в последующем при работе с экземплярами производных классов, через интерфейс базового, у нас будет вызываться одна из этих реализаций.

Именно благодаря полиморфизму программа, в процессе своей работы, может выбирать различные реализации при вызове методов с одним названием.

5.1 Объявление класса

Для объявления класса в Dart используется ключевое слово `class`. Для примера давайте опишем посредством класса один из объектов предметной области ветеринарной клиники с его состоянием и поведением – кота. Для этого создадим в новый файл «`cat.dart`»:

```
//cat.dart
class Cat{
  late final String name;
  int age = 0;
  bool sleepState = true;

  void sleep(){
    if(!sleepState){
      sleepState = true;
      print('Кот засыпает: Хр-р-р-р-р...');
    }
    else{
      print('Сон во сне... мmmm...');
    }
  }

  void wakeUp(){
    if(sleepState){
      sleepState = false;
      print('Лениво потягиваясь, открывает глаза...');
    }
  }

  void helloMaster(){
    if(!sleepState){
      print('Мя-я-я-я-у!!!');
    }
  }
}
```

```

void currentState(){
    if(sleepState){
        print('Кот спит');
    }
    else{
        print('Кот бодрствует');
    }
}
}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
    // создаем экземпляр класса Cat
    var cat = Cat()
        ..age=3
        ..name='Тимоха'
        ..sleepState=false;

    cat.helloMaster(); // Мя-я-я-я-у!!!
    cat.currentState(); // Кот бодрствует
    cat.sleepState = true;
    cat.currentState(); // Кот спит
    cat.sleep(); // Сон во сне... ммм...
}

```

В приведенном коде мы объявили класс `Cat`, описывающий состояние и поведения объекта кот. По умолчанию в Dart все переменные и методы класса являются `public` (публичными) и именно поэтому мы можем к ним обращаться и менять их значения из клиентского кода (в нашем случае – функция `main`). В том случае, если нам необходимо объявить приватную переменную, либо метод, то их имя должно начинаться с символа нижнего подчеркивания «`_`». К таким методам мы можем обращаться через публичные методы, а для доступа к приватным переменным, чтобы получить или установить их значения принято использовать геттеры (`get`) и сеттеры (`set`).

Давайте перепишем класс `Cat` с учетом того, что переменная состояния сна и бодрствования `sleepState` теперь будет приватной и

класс не предоставляет возможность для установки или чтения её значений:

```
//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  void sleep(){
    if(!_sleepState){
      _sleepState = true;
      print('Кот засыпает: Хр-р-р-р-р...');
    }
    else{
      print('Сон во сне... мmmm...');
    }
  }

  void wakeUp(){
    if(_sleepState){
      _sleepState = false;
      print('Лениво потягиваясь, открывает глаза...');
    }
  }

  void helloMaster(){
    if(!_sleepState){
      print('Мя-я-я-я-у!!!');
    }
  }

  void currentState(){
    if(_sleepState){
      print('Кот спит');
    }
    else{
      print('Кот бодрствует');
    }
  }
}
```

```

}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
  // создаем экземпляр класса Cat
  var cat = Cat()
    ..age=3
    ..name='Тимоха';

  /*
  // ошибка
  var cat = Cat()
    ..age=3
    ..name='Тимоха'
    ..sleepState=true;
  */

  cat.helloMaster(); // Мя-я-я-я-у!!!
  cat.currentState(); // Кот бодрствует
  cat.sleep(); // Кот засыпает: Хр-р-р-р-р...
  cat.sleep(); // Сон во сне... мmmm...
  cat.wakeUp(); // Лениво потягиваясь, открывает глаза...
}

```

Теперь из модуля, где объявлена функция `main` нельзя обращаться к приватной переменной класса `Cat` - `_sleepState`. Но это действительно только в том случае, если описываемый класс импортируется. То есть символ «`_`» скрывает переменные и методы класса от других элементов приложения только в случае его импортирования. В рамках того модуля (dart-файла), где описывается класс с его приватными переменными они остаются общедоступными. Для примера перенесем класс `Cat`, в котором имеется приватная переменная из файла «`cat.dart`» в файл с функцией верхнего уровня «`main`» и изменим из него значение переменной `_sleepState` экземпляра класса `Cat`:

```

//main.dart
class Cat{
// без изменений относительно предыдущего примера
}

void main(List<String> arguments) {
// создаем экземпляр класса Cat
var cat = Cat()
  ..age=3
  ..name='Тимоха'
  .._sleepState=false;

cat.helloMaster(); // Мя-я-я-я-у!!!
cat.currentState(); // Кот бодрствует
cat._sleepState = true;
cat.currentState(); // Кот спит
cat.sleep(); // Сон во сне... ммм...
cat._sleepState = false;
cat.currentState(); // Кот бодрствует
}

```

В связи с такой особенностью поведения приватных методов и переменных необходимо помнить о том, что относительно того модуля (файла), где они были объявлены, что переменные класса или методы, что приватные переменные модуля или функции, остаются публичными.

Вернем класс в файл «cat.dart» и напомним геттер и сеттер для установки и чтения значения переменной - `_sleepState`:

```

//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  bool get isSleep => _sleepState;
  set setSleepState(bool val) => _sleepState = val;

  // остальные методы не изменялись
}

```

```

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
  // создаем экземпляр класса Cat
  var cat = Cat()
    ..age=3
    ..name='Тимоха'
    ..setSleepState=false;

  cat.helloMaster(); // Мя-я-я-я-у!!!
  cat.currentState(); // Кот бодрствует
  print(cat.isSleep);
  cat.setSleepState = true;
  cat.currentState(); // Кот спит
  cat.sleep(); // Сон во сне... мммм...
}

```

5.2 Конструктор класса

Посредством конструктора создается экземпляр класса и когда он не объявляется явно, то компилятор создает конструктор по умолчанию, который не принимает на свой вход аргументы для инициализации состояния создаваемого экземпляра класса.

Давайте рассмотрим несколько способов объявления конструктора класса в Dart. В первом случае мы можем использовать стандартное объявление конструктора из языков программирования семейства Си:

```

//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  Cat(String name, int age) {
    this.name = name;
    this.age = age;
  }

  bool get isSleep => _sleepState;
}

```

```

    set setSleepState(bool val) => _sleepState = val;

    // остальные методы не изменялись
}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
    // создаем экземпляр класса Cat
    var cat = Cat('Тимоха', 3);
    print(cat.name); // Тимоха
    cat.helloMaster(); // Мя-я-я-я-у!!!
    cat.currentState(); // Кот бодрствует
}

```

Так как имена аргументов конструктора и переменных класса идентичны, то мы должны явным образом, через ссылку на текущий экземпляр класса - `this`, указать какой переменной значение какого аргумента конструктора присваивается. Если названия имен переменных класса и аргументов конструктора отличаются, то конструктор можно объявить следующим образом:

```

//cat.dart
class Cat{
    late final String name;
    int age = 0;
    // по умолчанию кот, при создании экземпляра класса
    // всегда бодрствует
    bool _sleepState = false;

    Cat(String n, int i) {
        name = n;
        age = i;
    }

    bool get isSleep => _sleepState;
    set setSleepState(bool val) => _sleepState = val;

    // остальные методы не изменялись
}

```

Еще один способ объявления конструктора позволяет его записать более компактно. В этом случае производится явное связывание имен аргументов конструктора с переменными экземпляра создаваемого класса:

```
//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  Cat(this.name, this.age);

  bool get isSleep => _sleepState;
  set setSleepState(bool val) => _sleepState = val;

  // остальные методы не изменялись
}
```

Аналогичным образом мы можем через конструктор инициализировать значения приватных переменных экземпляра класса:

```
//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  Cat(this.name, this.age, this._sleepState);
  // остальные методы не изменялись
}
```

```
//main.dart
import 'cat.dart';

void main(List<String> arguments) {
  // создаем экземпляр класса Cat
  var cat = Cat('Тимоха', 3, true);
}
```



```

    print(cat.name); // Тимоха
    cat.helloMaster(); // Мя-я-я-я-у!!!
    cat.currentState(); // Кот спит
}

```

К конструкторам и методам класса применимы все те же способы передачи аргументов, что и к функциям. Для примера сделаем передачу в конструктор флага бодрствования кота и его возраста не обязательными:

```

//cat.dart
class Cat{
    late final String name;
    int age = 0;
    // по умолчанию кот, при создании экземпляра класса
    // всегда бодрствует
    bool _sleepState = false;

    Cat(this.name, [this.age=4, this._sleepState=true]);
    // остальные методы не изменялись
}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
    // создаем экземпляр класса Cat
    var cat = Cat('Тимоха');
    print('Возраст кота "${cat.name}" = ${cat.age}');
    // Возраст кота "Тимоха" = 4
    cat.helloMaster(); // Мя-я-я-я-у!!!
    cat.currentState(); // Кот спит
}

```

Теперь, вместо позиционных используем именованные аргументы. Так как их имя не может начинаться с символа нижнего подчеркивания, необходимо будет изменить имя аргумента, который отвечает за передачу флага бодрствования кота и явно указать, что приватная переменная инициализируется значением этого аргумента конструктора:

```

//cat.dart
class Cat{

```

```

late final String name;
int age = 0;
// по умолчанию кот, при создании экземпляра класса
// всегда бодрствует
bool _sleepState = false;

Cat({required this.name, this.age = 2, required bool sleepState})
  : _sleepState = sleepState;
// остальные методы не изменялись
}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
// создаем экземпляр класса Cat
var cat = Cat(name: 'Тимоха', sleepState: true);
print('Возраст кота "${cat.name}" = ${cat.age}');
// Возраст кота "Тимоха" = 2
cat.helloMaster(); // Мя-я-я-я-у!!!
cat.currentState(); // Кот спит
}

```

В отличие от С-образных языков программирования Dart не поддерживает перегрузку конструктора, то есть мы не можем объявить несколько конструкторов с одним именем, но принимающие на вход различное количество аргументов или отличающиеся типом принимаемых аргументов. Вместо механизма перегрузки конструкторов программисту предоставляется возможность объявлять именованный конструктор:

```

//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  Cat({required this.name, this.age = 2, required bool sleepState})
    : _sleepState = sleepState;
}

```

```

    Cat.onlyName(this.name);

    Cat.fromNameAndAge(this.name, int old): age = old;
    // остальные методы не изменялись
}

//main.dart
import 'cat.dart';

void catProcessing(Cat cat){
    print('Возраст кота "${cat.name}" = ${cat.age}');
    cat.helloMaster();
    cat.currentState();
}

void main(List<String> arguments) {
    // создаем экземпляр класса Cat
    var cat = Cat(name: 'Тимоха', sleepState: true);
    catProcessing(cat);
    print('*'*20);
    cat = Cat.onlyName('Тимоха');
    catProcessing(cat);
    print('*'*20);
    cat = Cat.fromNameAndAge('Тимоха', 1);
    catProcessing(cat);
}

/*
Возраст кота "Тимоха" = 2
Кот спит
*****
Возраст кота "Тимоха" = 0
Мя-я-я-я-у!!!
Кот бодрствует
*****
Возраст кота "Тимоха" = 1
Мя-я-я-я-у!!!
Кот бодрствует
*/

```

Также имеется возможность из именованного конструктора вызывать основной конструктор класса:

```
//cat.dart
class Cat{
  late final String name;
  int age = 0;
  // по умолчанию кот, при создании экземпляра класса
  // всегда бодрствует
  bool _sleepState = false;

  Cat({required this.name, this.age = 2, required bool sleepState})
    : _sleepState = sleepState;

  Cat.onlyName(String nameCat)
    : this(name: nameCat, sleepState: true);

  Cat.fromNameAndAge(String name, int old)
    : this(name: name, age: old, sleepState: true);
  // остальные методы не изменялись
}

//main.dart
import 'cat.dart';

void catProcessing(Cat cat){
  print('Возраст кота "${cat.name}" = ${cat.age}');
  cat.helloMaster();
  cat.currentState();
}

void main(List<String> arguments) {
  // создаем экземпляр класса Cat
  var cat = Cat(name: 'Тимоха', sleepState: true);
  catProcessing(cat);
  print('*'*20);
  cat = Cat.onlyName('Тимоха');
  catProcessing(cat);
  print('*'*20);
  cat = Cat.fromNameAndAge('Тимоха', 1);
  catProcessing(cat);
}
```

```

/*
Возраст кота "Тимоха" = 2
Кот спит
*****

Возраст кота "Тимоха" = 2
Мя-я-я-я-у!!!
Кот бодрствует
*****

Возраст кота "Тимоха" = 1
Кот спит
*/

```

5.2.1 Константный конструктор

Его необходимо использовать в тех случаях, когда значения объектов и переменных класса не меняются, то есть все они объявлены через `final`. Для использования такого объявления конструктора, перед ним следует добавить ключевое слово `const`:

```

//cat.dart
class ImmutableCat {
  final String name;
  final int age;

  const ImmutableCat(this.name, this.age);

  void helloMaster(){
    print('Мя-я-я-я-у!!!');
  }
}

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
  var cat = const ImmutableCat('Тимоха', 3);
  var newcat = const ImmutableCat('Тимоха', 3);

  var barsik = const ImmutableCat('Барсик', 2);
}

```

```

    print(identical(cat, newcat)); // cat == newcat
    print(identical(cat, barsik)); // cat != barsik
    barsik.helloMaster(); // Мя-я-я-я-у!!!
}

```

Переменная `cat` и `newcat` хранят ссылку на один и тот же объект, это связано с тем, что при их создании использовались идентичные аргументы для инициализации экземпляра класса. При этом, необходимо не забывать при создании экземпляра класса перед его конструктором использовать ключевое слово `const`, иначе будет создаваться новый экземпляр класса, даже в том случае, когда производится инициализация одинаковыми значениями:

```

//main.dart
import 'cat.dart';

void main(List<String> arguments) {
    var cat = const ImmutableCat('Тимоха', 3);
    var newcat = ImmutableCat('Тимоха', 3);

    var barsik = const ImmutableCat('Барсик', 2);

    print(identical(cat, newcat)); // cat != newcat
    print(identical(cat, barsik)); // cat != barsik
}

```

5.2.2 Фабричный конструктор

Обычные конструкторы отвечают за создание и инициализацию новых экземпляров класса. А что, если нам необходимо, чтобы конструктор в случае создания объекта с параметрами, по которым уже ранее создавался объект, возвращал именно его? Или в системе присутствовал только один объект (экземпляр) нужного нам класса?

Для этих случаев необходимо использовать фабричные конструкторы, путем добавления перед конструктором класса ключевого слова `factory`. При это фабричный конструктор от обычных отличается еще тем, что в нем явно должен осуществляться возврат экземпляра класса посредством `return`.

Для начала приведем пример, когда в разрабатываемой программной системе необходимо обеспечить наличие всего лишь

одного экземпляра класса (паттерн Singleton/Одиночка). Представим, что у нас в квартире может существовать только один кот. Он полноправный хозяин в квартире и не потерпит наличие конкуренции:

```
// singlecat.dart
class SingleCat{
  String _name;
  int age;
  static SingleCat _singleCat = SingleCat.fromName('',0);

  SingleCat.fromName(this._name, this.age);

  factory SingleCat(String name, int age){
    if(_singleCat._name == ''){
      _singleCat = SingleCat.fromName(name, age);
      print('Создаем экземпляр класса кота');
    }
    else{
      print('Экземпляр класса кота был создан ранее!');
    }
    return _singleCat;
  }
  String get name => _name;
}
//main.dart
import 'singlecat.dart';

void main(List<String> arguments) {
  var cat = SingleCat('Тимоха', 2);
  var newCat = SingleCat('Барсик', 4);
  print(newCat.name);
}
/*
Создаем экземпляр класса кота
Экземпляр класса кота был создан ранее!
Тимоха
*/
```

Теперь же представим ситуацию, что у нас в системе огромное количество книг, и чтобы не создавать дубликаты, мы в случае запроса на создания экземпляра класса с параметрами, по которым был ранее создан экземпляр класса книги, будем возвращать его. А если экземпляр

класса книги с необходимыми параметрами не создавался, то мы его создадим, сделаем себе об этом заметку и вернем в клиентский код:

```
class Book{
    final String name;
    final int pages;
    static var _booksMap = <String, Book>{};

    Book.fromSettings(this.name, this.pages);

    factory Book(String name, int pages){
        var cache = name.toLowerCase() + pages.toString();
        return _booksMap.putIfAbsent(cache,
            () => Book.fromSettings(name, pages));
    }
}

void main(List<String> arguments) {
    var book1 = Book('Война и Мир т.1', 1234);
    var book2 = Book('Тихий Дон т.1', 400);
    var book3 = Book('Евгений Онегин', 250);
    var book4 = Book('Война и Мир т.1', 1234);
    print(identical(book2, book3)); // false
    print(identical(book1, book4)); // true
}
```

5.3 Статические переменные и методы класса

В предыдущих примерах использовались статические переменные класса. Их отличие от обычных заключается в том, что они будут хранить одно и то же значение вне зависимости от того, с каким экземпляром класса сейчас производится работа. Также к ним можно обращаться только через имя самого класса (без создания экземпляра класса), либо прописав сеттеры и геттеры. Отдельно стоит обратить внимание на то, что статическая переменная должна быть инициализирована до момента её использования (обращения к ней):

```
class Book{
    static var bookPages = 10;

    int get pages => bookPages;
}
```



```

void main(List<String> arguments) {
  var book1 = Book();
  print(book1.pages); // 10
  Book.bookPages = 20; // меняем значение
  var book2 = Book();
  print(book2.pages); // 20
}

```

Статические методы можно вызывать, обращаясь к ним через имя класса, без создания самого экземпляра класса или написать отдельную для экземпляра класса, которая будет переадресовывать вызов статическому методу:

```

class Calc{
  static int add(int a, int b){
    return a + b;
  }

  int sum(int a, int b) => add(a, b);
}

```

```

void main(List<String> arguments) {
  print(Calc.add(3, 5)); // 8
  print(Calc.add(13, 5)); // 18
  var calc = Calc();
  print(calc.sum(13, 5)); // 18
}

```

5.4 Перегрузка операторов

Dart предоставляет возможность программисту перегружать стандартные операторы (см. главу 2), тем самым создавая более гибкие классы. Для примера рассмотрим перегрузку оператора сложения на следующем примере: когда выполняется сложение между двумя экземплярами классов книги будем их упаковывать в коробку, с которой можно продолжить работу в клиентском коде.

```

class Box {
  List<Book> items;
  Box(this.items);
}

```

```

void printBooks(){
    items.forEach((element) {
        print(element.name);
    });
}

void operator +(Book book){
    items.add(book);
}

class Book{
    final String name;
    final int pages;

    Book(this.name, this.pages);

    Box operator +(Book otherBook){
        return Box([this, otherBook]);
    }
}

void main(List<String> arguments) {
    var book1 = Book('Война и Мир т.1', 1234);
    var book2 = Book('Тихий Дон т.1', 400);
    var box = book1 + book2;
    box.printBooks();
    print('-' * 30);
    box+Book('Евгений Онегин', 250);
    box.printBooks();
}
/*
Война и Мир т.1
Тихий Дон т.1
-----
Война и Мир т.1
Тихий Дон т.1
Евгений Онегин
*/

```

5.5 Наследование и переопределение методов

В Dart нет множественного наследования, то есть наследоваться можно только от одного базового класса. В тоже самое время класс может реализовывать множество интерфейсов (тема следующего раздела). Для примера давайте заведем еще один класс животного – **Dog** (собака), а общие состояния и поведение, присущие коту и собаке выделим в базовый класс **Animal** (Животное) от которого будем наследоваться, используя ключевое слово **extends**. После чего в основной программе заведем список **Animal**, куда будем добавлять экземпляры производных классов, приводя их к базовому.

```
//animal.dart
class Animal {
  final String name;
  int age;
  bool sleepState = false;

  Animal(this.name, this.age);

  void sleep(){
    if(!sleepState){
      sleepState = true;
      print('$name засыпает: Хр-р-р-р-р...');
    }
    else{
      print('Сон во сне... мmmm...');
    }
  }

  void wakeUp(){
    if(sleepState){
      sleepState = false;
      print('Лениво потягиваясь, открывает глаза...');
    }
  }

  void helloMaster(){
    print('$name: o_O');
  }

  void currentState(){
```

```

    if(sleepState){
        print('$name спит');
    }
    else{
        print('$name бодрствует');
    }
}
}
}

```

//cat.dart

```
import 'animal.dart';
```

```
class Cat extends Animal {
```

```
    Cat(String name, int age) : super(name, age);
```

```
    @override //переопределяем метод базового класса
```

```
    void helloMaster(){
```

```
        if(!sleepState){
```

```
            print('$name: Мя-я-я-я-у!!!');
```

```
        }
```

```
    }
```

```
    void purr(){ //метод доступный только экземплярам класса Cat
```

```
        print('$name: М-р-р-р-р-р');
```

```
    }
```

```
}
```

//dog.dart

```
import 'animal.dart';
```

```
class Dog extends Animal {
```

```
    late int _aggressionLevel;
```

```
    Dog(String name, int age, int aggressionLevel) : super(name, age) {
```

```
        _aggressionLevel = aggressionLevel;
```

```
    }
```

```
    @override //переопределяем метод базового класса
```

```
    void helloMaster() {
```

```
        if (!sleepState) {
```

```
            var hello = 'У-у-у-у!';
```

```

    if(_aggressionLevel >= 5){
        hello = 'Гр-р-р-р-р!!!';
    }
    else if (_aggressionLevel > 2){
        hello = 'Гав!';
    }
    print('$name: $hello');
}
}

//метод доступный только экземплярам класса Dog
void whine() => print('$name: У-у-у-у-у!');
}

```

//main.dart

```

import 'animal.dart';
import 'cat.dart';
import 'dog.dart';

void main(List<String> arguments) {
    var animalsList = <Animal>[];
    animalsList.add(Cat('Муся', 4));
    animalsList.add(Cat('Тимоха', 5));
    animalsList.add(Dog('Барсик', 5, 6));
    animalsList.add(Dog('Барбос', 2, 3));
    animalsList.add(Dog('Рекс', 3, 1));
    animalsList.add(Animal('Ттикачу', 33));

    animalsList[0].sleep(); // Муся засыпает: Хр-р-р-р-р...
    animalsList[3].sleep(); // Барбос засыпает: Хр-р-р-р-р...

    print('* * 30);
    for(var it in animalsList){
        it.helloMaster();
    }
    print('* * 30);
    for(var it in animalsList){
        if (it is Dog){
            // теперь у it доступны методы класса Dog
            print('${it.name} - собака');
            it.whine();
        }
    }
}

```

```

    }
    else if (it is Cat){
        // теперь у it доступны методы класса Cat
        print('${it.name} - кот');
        it.purr();
    }
    else{
        print('${it.name}? Это что еще за покемон???');
    }
    print('- ' * 30);
}
}

/*
*****
Тимоха: Мя-я-я-я-у!!!
Барсик: Гр-р-р-р-р!!!
Рекс: У-у-у-у!
Пикачу: о_О
*****

Муся - кот
Муся: М-р-р-р-р-р
-----
Тимоха - кот
Тимоха: М-р-р-р-р-р
-----
Барсик - собака
Барсик: У-у-у-у-у!
-----
Барбос - собака
Барбос: У-у-у-у-у!
-----
Рекс - собака
Рекс: У-у-у-у-у!
-----
Пикачу? Это что еще за покемон???
-----
*/

```

В приведенном примере осуществляется работа с экземплярами производных классов `Cat` и `Dog` через интерфейс базового класса `Animal` (за исключением последнего элемента списка). Говоря другими словами - мы привели производные классы к базовому. Это позволяет нам абстрагироваться от того, с каким именно животным мы сейчас работаем

и вместо двух списков (отдельно для котов, отдельно для собак) держать в системе всего один список из приведенных к базовому, производных классов. Так как у базового класса не было нормальной реализации метода `helloMaster`, мы его переопределили посредством декоратора `@override`, после чего в каждом из производных классов написали специфичную только для него реализацию. Это позволяет после приведения производного класса к базовому, вызывая метод определенный в базовом классе на самом деле вызывать переопределенную в производном классе его реализацию.

Если же из переопределенного метода необходимо обратиться к исходному методу базового класса, это делается через ключевое слово `super`, которое позволяет нам работать с поведением и состоянием базового класса.

5.6 Абстрактный класс и интерфейс

Не всегда базовый класс может содержать реализации всех объявленных в нем методов. Так как такие классы представляют из себя некоторое обобщение, то часть реализации их методов может отдаваться на откуп производным классам. То есть базовый класс можно рассматривать как некоторый обобщенный интерфейс, через который мы в клиентском коде можем работать с экземплярами производных классов, приводя их к базовому. Это дает нам еще один уровень инкапсуляции. Так, например, после того как привели производный класс к базовому мы можем вызывать только методы базового класса, так как методы, специфичные для производного класса становятся не доступны. Если же нам нужно вызвать один из таких методов производного класса, то сперва необходимо выполнить проверку можно ли текущий экземпляр класса, с которым осуществляется работа, привести к необходимому производному классу.

Методы, которые объявлены, но не имеют реализации, называются **чисто виртуальными методами**. А класс, который содержит хоть один виртуальный метод – **абстрактный класс**. Отличие абстрактного класса от обычного заключается в том, что экземпляр абстрактного класса не может быть создан. Но к таким классам мы можем приводить производные от них классы. То есть что базовый, что абстрактный класс может выступать в роли публичного интерфейса к экземплярам

производных от них классов. В отличие от C++ в Dart только у класса, который явно объявлен как абстрактный (**abstract**), можно указывать методы без описания их реализации. При этом наличие таких методов в самом абстрактном классе не обязательно.

Давайте перепишем класс **Animal** таким образом, чтобы он стал абстрактным базовым классом:

```
//animal.dart
abstract class Animal {
  final String name;
  int age;
  bool sleepState = false;

  Animal(this.name, this.age);

  void helloMaster(); // чисто виртуальный метод
  // остальные методы не изменялись
}
```

По поводу интерфейса принято говорить, что «класс реализует такой-то интерфейс». Его отличие от абстрактного или обычного класса заключается в том, что определенные в нем состояние и поведение должны быть также реализованы в том классе, который его реализует. В случае наследования нам не надо было прописывать все переменные и методы базового класса в производном заново, а только те методы, которые хотели переопределить. Здесь же всё - наоборот. Все переменные и методы, объявленные в интерфейсе, должны быть объявлены и иметь реализацию в том классе, который этот интерфейс реализует.

Таким образом, если в случае с наследованием мы наследуем состояние и поведение, то в случае с интерфейсом – объявляем контракт, которому должен следовать класс реализующий интерфейс.

Каждый класс в Dart неявно определяет интерфейс и когда нам необходимо, чтобы разрабатываемый класс А поддерживал API другого класса В, не наследуя его реализацию, то класс А должен реализовать интерфейс В.

В отличие от наследования, класс может реализовывать сколько угодно интерфейсов. Для указания того, что текущий класс реализует интерфейс другого класса используется ключевое слово **implements**.

В качестве примера рассмотрим ситуацию, когда у нас есть коробка для хранения вещей и шкаф. Оба этих объекта представляют собой систему хранения. Мы можем добавлять в них вещи, забирать последнюю добавленную вещь и считать общую сумму веса тех вещей, которые в них хранятся. Так как это объекты из различной предметной области, то они не будут наследоваться от базового абстрактного класса СистемаХранения, но будут реализовывать его интерфейс:

```
class Item{
    final String name;
    final double weight;

    Item(this.name, this.weight) ;
}

abstract class StorageSystem {
    void addItem(Item item);

    Item popItem();

    double systemWeight();
}

class Box implements StorageSystem {
    var itemList = <Item>[];
    final double weightLimit;

    Box(this.weightLimit);

    @override
    void addItem(Item item) {
        var currentSystemWeight = systemWeight();
        if((currentSystemWeight+item.weight) < weightLimit){
            itemList.add(item);
            print('${item.name} добавлен(о/а) в коробку!');
        }
        else{
            print('${item.name} не помещается в коробку!');
        }
    }
}
```

```

@Override
Item popItem() {
    return itemsList.removeLast();
}

@Override
double systemWeight() {
    var sum = 0.0;
    itemsList.forEach((element) {
        sum += element.weight;
    });
    return sum;
}
// методы, характерные для коробки
}

class Cupboard implements StorageSystem {
    var itemsList = <Item>[];

    @Override
    void addItem(Item item) {
        itemsList.add(item);
        print('${item.name} добавлен(о/а) в шкаф!');
    }

    @Override
    Item popItem() {
        return itemsList.removeLast();
    }

    @Override
    double systemWeight() {
        var sum = 0.0;
        itemsList.forEach((element) {
            sum += element.weight;
        });
        return sum;
    }
    // методы, характерные для шкафа
}

```

```

void main(List<String> arguments) {
    var box = Box(18);
    var cupboard = Cupboard();
    StorageSystem? storageSystem = box;
    storageSystem.addItem(Item('Книга', 2.6));
    storageSystem.addItem(Item('Чайник', 3.9));
    storageSystem.addItem(Item('Гантеля', 10));
    storageSystem.addItem(Item('Монитор', 4));

    print(storageSystem.popItem().name);
    print(storageSystem.systemWeight());

    storageSystem = cupboard;
    print(storageSystem.systemWeight());
    storageSystem.addItem(Item('Монитор', 4));
    storageSystem.addItem(Item('Чайник', 3.9));
}

```

Как видно из примера, посредством приведения к интерфейсу мы можем работать с любой системой хранения, которая реализует этот интерфейс и нам не надо прописывать в функции `main` код для добавления, изъятия или подсчета веса хранимых вещей для каждого из классов, реализующих этот интерфейс. Достаточно привести экземпляр класса к интерфейсу и работать через него. Таким образом мы можем достаточно просто переключаться между шкафом или коробкой и если добавится еще один объект, реализующий интерфейс системы хранения, не возникнет никаких трудностей при взаимодействии с ним из клиентского кода.

В качестве интерфейса не обязательно использовать абстрактный класс, им может выступать и обычный:

```

class Person {
    final String _name;
    int age;
    Person(this._name, this.age);

    int howMuchOlder(Person person) => age - person.age;
    String greet(Person person){
        return 'Привет, ${person._name}!!!. Меня зовут $_name.';
    }
}

```

```

class Alan implements Person{
    @override
    int age = 33;

    @override
    String get _name => 'Алан';

    @override
    String greet(Person person) {
        return 'Привет, ${person._name}!!! Меня зовут $_name.';
    }

    @override
    int howMuchOlder(Person person) {
        return age*2 - person.age;
    }
}

class Impostor implements Person {
    @override
    int age = 0;

    @override
    String get _name => '';

    @override
    String greet(Person person) {
        return 'Вот ты и попался, ${person._name}!!!';
    }

    @override
    int howMuchOlder(Person person) {
        return -1;
    }
}

String greet(Person firstperson, Person secondPerson){
    return firstperson.greet(secondPerson);
}

```

```

int checkAge(Person firstperson, Person secondPerson){
    return firstperson.howMuchOlder(secondPerson);
}

void main(List<String> arguments) {
    var maxim = Person('Макс', 45);
    var alan = Alan();
    var impostor = Impostor();

    print(greet(maxim, alan)); // Привет, Алан!!!. Меня зовут Макс.
    print(greet(impostor, maxim)); // Вот ты и попался, Макс!!!
    print(checkAge(maxim, alan)); // 12
    print(checkAge(alan, maxim)); // 22
}

```

5.7 Mixins (Примеси)

В объектно-ориентированном программировании помимо обычных, абстрактных классов и интерфейсов, существует еще такой «зверь», как mixins (примеси).

Давайте представим, что у нас есть два класса животных, каждый из которых обладает собственным набором поведения. Самое первое, что придет на ум – выделить общее состояние и поведение в базовый класс. Пока в системе два класса животных – всё работает отлично, но вот нам понадобился класс птицы. Казалось бы, у базового класса животных и птицы много общего, но вот летать животные не умеют, хотя также бегают, едят, купаются и т. д., у них имеется атрибут возраста и названия вида, к которому принадлежат. Очевидно, что хочется найти наиболее эффективную структуру для написания этих классов с максимальной возможностью повторного использования кода. Тут на помощь и приходят примеси, поскольку они позволяют вставлять блоки кода в класс, без необходимости создания производного класса. Для того, чтобы объявить примесь, следует использовать ключевое слово `mixin`:

```

mixin Eating{
    void eat(){
        print('Кушать');
    }
}

```

```

mixin Running{
  void run(){
    print('Бежать');
  }
}

abstract class Purring{
  void purr(){
    print('Му-р-р-р-р');
  }
}

abstract class Whining{
  void whine(){
    print('У-у-у-у-у');
  }
}

abstract class flying {
  void fly() {
    print('Полет');
  }
}

class Cat extends Purring with Eating, Running {
  void goToEat(){
    run();
    eat();
    purr();
    print('От послеобеденного мурчания кота можно оглохнуть!');
  }
}

class Dog extends Whining with Eating, Running {
  void runToEat(){
    run();
    eat();
    print('Пес полакомился обедом');
  }
}

```

```

class Bird extends flying with Eating, Running {
    void flyToEat(){
        fly();
        run();
        eat();
        print('После обеда птица нежится на солнце');
    }
}

```

```

void main(List<String> arguments) {
    var bird = Bird()..flyToEat();
    print('-' * 30);
    var cat = Cat()..goToEat();
    print('-' * 30);
    var dog = Dog()..runToEat();

    print('-' * 30);
    // приводим экземпляры класса к примеси
    var listMixin = <Eating>[bird, cat, dog];
    listMixin.forEach((element) {
        element.eat();
    });
}
/*
Полет
Бежать
Кушать
После обеда птица нежится на солнце
-----
Бежать
Кушать
Му-р-р-р-р
От послеобеденного мурчания кота можно оглохнуть!
-----
Бежать
Кушать
Тес полакомился обедом
-----
Кушать
Кушать
Кушать
*/

```

А теперь представим, что после каждого приема пищи птица начинает щебетать. Что в этом случае делать? Ввести дополнительную примесь, подключить её и использовать в классе или дать возможность примеси вызывать методы класса птицы, то есть путем вызова метода щебетания запустить цепочку поиска еды и насыщения? У каждого из этих подходов есть свои плюсы и минусы.

Давайте рассмотрим, как можно из примеси вызывать методы класса:

```
mixin Chipping on Bird{
    void chip(){
        flyToEat();
        print('Чик-чирик!');
    }
}

class Sparrow extends Bird with Chipping{

}

void main(List<String> arguments) {
    var bird = Sparrow()..chip();
}
/*
Лететь
Бежать
Кушать
После обеда птица нежится на солнце
Чик-чирик!
*/
```

5.8 Generics (Обобщения)

Использование дженериков позволяет уменьшить дублирование кода. То есть у разработчика появляется возможность использовать единый интерфейс и реализацию для многих типов. Это часто бывает полезным, если отличие между создаваемыми классами заключается только в типе некоторых его переменных, принимаемых на вход типов аргументов методов или возвращаемых ими значений. Примером самых элементарных реализаций классов с использованием дженериков являются коллекции: `List<T>`, `Set<T>`, `Map<K, V>`.


```

class MyID {
    final int id;
    final String idString;

    MyID(this.id): idString = id.toString();

    @override
    String toString() {
        return idString;
    }
}

class Product<T>{
    T id;
    final String name;
    final double price;

    Product(this.id, this.name, this.price);

    T getId() => id;
    void setId(T idProduct) => id = idProduct;

    @override
    String toString() {
        return 'Продукт: $name с id: $id стоит $price тугриков';
    }
}

void main(List<String> arguments) {
    var product = Product<int>(0, 'Булочка', 33.5);
    print(product);
    var newProduct = Product<MyID>(MyID(10), 'Пирожок', 50);
    print(newProduct);
}
// Продукт: Булочка с id: 0 стоит 33.5 тугриков
// Продукт: Пирожок с id: 10 стоит 50.0 тугриков

```

Иногда нам нужно задать ограничение при использовании дженериков. Это делается для того, чтобы не все типы данных могли указываться программистом для создаваемых экземпляров классов. В таком случае необходимо явно задать ограничение и можно будет в

качестве дженерика передавать как указанный класс, так и его производные классы:

```
class Item{
    final String name;
    final int value;
    Item(this.name, this.value);
}

class Book extends Item{
    Book(String name, int pages): super(name, pages);

    Box operator +(Book otherBook){
        return Box([this, otherBook]);
    }
}

class Magazine extends Item{
    Magazine(String name, int pages): super(name, pages);
}

class Box<T extends Item> {
    List<T> items;
    Box(this.items);

    void printBooks(){
        items.forEach((element) {
            print(element.name);
        });
    }

    void operator +(T book){
        items.add(book);
    }
}

void main(List<String> arguments) {
    var book1 = Book('Война и Мир т.1', 1234);
    var book2 = Book('Тихий Дон т.1', 400);
    var box = book1 + book2;
    box.printBooks();
    print('-' * 30);
}
```

```

    box+ Magazine('Огонёк', 250);
    box.printBooks();
}
/*
Война и Мир т.1
Тихий Дон т.1
-----
Война и Мир т.1
Тихий Дон т.1
Огонёк
*/

```

Также дженерики можно использовать с функциями:

```

class MyID {
    final int id;
    final String idString;

    MyID(this.id): idString = id.toString();

    @override
    String toString() {
        return idString;
    }
}

T firstElement<T>(List<T> list){
    return list[0];
}

void main(List<String> arguments) {
    var list = <MyID>[MyID(2), MyID(33)];
    print(firstElement(list)); // 2
}

```

5.9 Enum (Перечисления)

Перечисления представляют собой особый вид классов, используемых для представления фиксированного числа постоянных значений. Для того, чтобы создать перечисление достаточно использовать ключевое слово **enum**. Каждому значению в перечислении соответствует свой целочисленный индекс. Например, первое значение имеет индекс 0, второе значение имеет индекс 1 и т. д.

```
enum State { none, open, close, lock}

void main(List<String> arguments) {
  print(State.none.index == 0); // true
  print(State.open.index == 1); // true
  // формируем список значений перечисления
  var listEnums = State.values;
  listEnums.forEach((element) {
    print('${element.index} => ${element.toString()}');
  });
}
/*
0 => State.none
1 => State.open
2 => State.close
3 => State.lock
*/
```

Чаще всего перечисления используют для создания машины состояний и выбора потока управления выполнения кода в приложении, посредством оператора **switch-case**, либо когда необходимо, чтобы одно из состояний объекта могло меняться только в четко определенном диапазоне возможных состояний.

Перечисления в Dart имеют ряд ограничений:

- нельзя создавать подклассы, примеси, объявления и реализации методов перечисления;
- нельзя явно создать экземпляр перечисления.

Резюме по разделу

В данном разделе рассмотрено что такое объектно-ориентированное программирование, какие принципы лежат в его основе и как они реализуются в Dart. Дополнительно затронуто понятие абстракции, абстрактного базового класса и интерфейса. Интерфейсы рекомендуется использовать, когда необходимо добавить уровень гибкости в разрабатываемое приложение, чтобы уменьшить связность более верхнего (абстрактного) уровня с нижними уровнями, в которых находится реализация функциональности. А также посредством интерфейсов можно предоставлять API для доступа к разрабатываемому модулю/компоненту/плагину и т. д. Поэтому понимание, что

представляет из себя интерфейс и как с ним работать – очень важно для программистов.

Дополнительно был рассмотрен такой инструмент, как дженерики (обобщения), который позволяет уменьшить дублирование кода. То есть у разработчика появляется возможность использовать единый интерфейс и реализацию для многих типов.

Вопросы для самопроверки

1. Что такое объект?
2. Как объект связан с классом?
3. Какие характеристики имеются у объекта?
4. Что такое состояние объекта? Как оно реализуется в классе?
5. Что такое поведение объекта? Как оно реализуется в классе?
6. Какие принципы лежат в основе ООП?
7. Как объявить класс?
8. Что такое экземпляр класса?
9. Для чего используется конструктор класса и как он объявляется?
10. Какие типы конструкторов класса существуют в Dart?
11. Что такое перегрузка оператора и для чего она используется?
12. Как объявляются и для чего используются статические переменные и методы класса?
13. Можно ли в Dart использовать множественное наследование? Какое ключевое слово используется для наследования от базового класса?
14. Для чего и как переопределяются методы базового класса в производном?
15. Что такое абстрактный класс и интерфейс? Чем они схожи, а в чем отличие?
16. Что такое mixins (примеси)? Для чего они используются?
17. Как в Dart реализованы generics (Обобщения)? Для чего они нужны?
18. Перечислите ограничения, которые имеются у `enum` (перечислений).

Упражнения

1. Реализуйте базовый класс «Машина» и несколько производных классов «Жигули» и «Ока», переопределите ряд методов и осуществите работу с экземплярами производных классов через интерфейс базового.

2. Реализуйте класс «Кружка» и «Человек». Дайте возможность человеку пить из кружки.
3. Реализуйте класс «Шкаф», в состав которого входит несколько систем хранения. У пользователя должна быть возможность поместить вещи в шкаф и забрать их оттуда.
4. Реализуйте класс «Гриф» и «Блин». На гриф с левой и с правой стороны можно навешивать блины. В конструкторе класса «Гриф» передавайте параметр максимальной загрузки грифа по весу.
5. Реализуйте класс для конвертации денег из одной валюты в другую.
6. Реализуйте модуль, в котором через единый интерфейс осуществляется работа с различным форматом файлов (txt, json, csv).
7. Посредством дженериков реализуйте класс «Гараж», где можно хранить различные объекты.
8. Напишите класс, в котором перегружены все базовые арифметические операции и класс. Проверьте его работу.
9. Напишите класс «Автомобиль», для таких состояний, как: стоп, ехать, повернуть, используйте перечисления.
10. Напишите базовый класс «Геометрическая фигура» и несколько производных классов (прямоугольник, треугольник, окружность и т. д.).
11. Напишите класс для перевода значения числа из одной системы счисления (десятичная, шестнадцатеричная и восьмеричная) в другую.
12. Напишите класс, который содержит список с геометрическими фигурами и позволяет находить фигуру с максимальной площадью.
13. Напишите класс для вычисления корней квадратного уравнения.
14. Напишите класс для поиска наименьшего общего делителя.
15. Напишите класс «Стол», на который можно ставить различные столовые приборы (производные классы от базового) и убирать их со стола.

6 Exceptions (Исключения)

В процессе работы приложения могут происходить непредвиденные ситуации: обращение по несуществующему индексу списка, деление на ноль и т. д. Таким образом, исключения – это ошибки, которые возникли в ходе выполнения кода и указывающие на имеющиеся проблемы в логике работы программы. Обычно при генерации исключения происходит экстренное завершение программы.

Несмотря на то, что исключения можно перехватывать и обрабатывать, тем самым оставляя программу в рабочем состоянии в ряде случаев это делать не стоит. Лучше пусть программа «упадет» и укажет вам на наличие ошибок, чем продолжит работать с некорректными данными. Особенно не стоит перехватывать для обработки исключений класс `Exception`, так как он является базовым классом для всех исключений, то есть какое бы не сгенерировалось исключение – оно будет обработано и если в блоке обработки не предусмотрена повторная генерация исключения, чтобы предупредить более верхний уровень приложения, вы можете и не узнать о наличии ошибки.

Для работы с исключениями Dart предоставляет для разработчиков такие классы, как `Exception` и `Error`, а также множество их производных классов.

6.1 Конструкция `try...catch...finally`

Общий вид данной конструкции можно записать следующим образом:

```
try{  
  // блок кода, который может генерировать исключение  
}  
catch(e){  
  // блок обработки исключения  
}  
finally{  
  // блок кода, который выполнится в любом случае,  
  // как при отсутствии исключения  
  // так и после его обработки  
}
```

Та часть кода, в которой может быть сгенерировано исключение, помещается в блок `try`. Если в ходе выполнения кода, помещенного в блок `try` было сгенерировано исключение, оно может быть перехвачено и обработано в блоке `catch`. При этом мы можем перехватывать, как высокоуровневое исключение (базовое `Exception`), так и специализированное, явно задавая тип перехватываемого исключения. Блок `finally` выполнится в любом случае, даже если будет сгенерировано исключение. Обычно в него помещают код, где завершается работа с файлом, сокетом и прочими объектами. То есть такой подход гарантирует, что при отсутствии или наличии исключения мы завершим работу с объектом, источником информации и т. д.

Давайте представим ситуацию, что в разрабатываемом нами приложении осуществляется целочисленное деление на значение, которое ввел пользователь и мы забыли на этапе ввода проверить, чтобы оно не равнялось нулю. В этом случае, в процессе работы приложения, будет сгенерировано исключение `IntegerDivisionByZeroException` и оно прекратит свою работу. Чтобы этого избежать, необходимо перехватить генерируемое исключение одним из следующих способов:

```
void main(List<String> arguments) {
    var inputValue = 0;
    var resultValue = 0;
    var scalingValue = 100;

    // 1
    try{
        resultValue = scalingValue ~/ inputValue;
    }
    catch(e){
        // перехват всех исключений
        print('Произошло деление на ноль!!!');
        print(e);
    }

    // Произошло деление на ноль!!!
    // IntegerDivisionByZeroException

    //2
    try{
        resultValue = scalingValue ~/ inputValue;
```



```

    } on IntegerDivisionByZeroException{
        // перехват специализированного исключения
        print('Произошло деление на ноль!!!');
    }
    // Произошло деление на ноль!!!

//3
    try{
        resultValue = scalingValue ~/ inputValue;
    } on Exception catch (e) {
        // Перехватывает все исключения
        print('Сгенерированное исключение: $e');
    } catch (e) {
        // Перехватывает вообще всё
        // в данном случае - ошибки
        print('Что-то из ряда вон выходящего: $e');
    }
    //Сгенерированное исключение: IntegerDivisionByZeroException

//4
    try{
        resultValue = scalingValue ~/ inputValue;
    } on IntegerDivisionByZeroException{
        // перехват специализированного исключения
        print('Произошло деление на ноль!!!');
    }
    finally{
        print('Что бы ни произошло, я - великолепен!!!');
    }
    // Произошло деление на ноль!!!
    // Что бы ни произошло, я - великолепен!!!
}

```

Как видно из примера, мы можем перехватывать одно, несколько или сразу все исключения. В том же случае, если сгенерированное исключение не соответствует ни одному из перехватываемых, то оно распространиться после завершения блока `finally`:

```

void main(List<String> arguments) {
    var inputValue = 0;
    var resultValue = 0;
    var scalingValue = 100;
    try{

```

```

    resultValue = scalingValue ~/ inputValue;
}
finally{
    print('Что бы ни произошло, я - великолепен!!!');
}
}
/*
Что бы ни произошло, я - великолепен!!!
Unhandled exception:
IntegerDivisionByZeroException
*/

```

6.2 Генерация исключений и ошибок

В тех случаях, когда нам самим необходимо генерировать сообщения об ошибках или исключения, следует использовать ключевое слово `throw`:

```

void main(List<String> arguments) {
    var inputValue = 0;
    var resultValue = 0;
    var scalingValue = 100;
    try{
        if (inputValue == 0){
            throw IntegerDivisionByZeroException();
        }
        resultValue = scalingValue ~/ inputValue;
    } on IntegerDivisionByZeroException{
        print('Произошло деление на ноль!!!');
    }
    catch(e){
        print('Ошибка: $e');
    }
}
// Произошло деление на ноль!!!

try{
    if (inputValue == 0){
        throw ArgumentError();
    }
    resultValue = scalingValue ~/ inputValue;
} on IntegerDivisionByZeroException{
    print('Произошло деление на ноль!!!');
}

```

```

catch(e){
    print('Ошибка: $e');
}
//Ошибка: Invalid argument(s)
}

```

Если генерируемое исключение не может быть обработано на данном уровне, может быть обработано частично или нам необходимо оповестить более верхний уровень приложения о сгенерированном исключении, его можно распространить за текущий блок обработки, используя ключевое слово **rethrow**:

```

void main(List<String> arguments) {
    var inputValue = 0;
    var resultValue = 0;
    var scalingValue = 100;
    try{
        resultValue = scalingValue ~/ inputValue;
    }on IntegerDivisionByZeroException{
        print('Произошло деление на ноль!!!');
        rethrow;
    } on ArgumentError catch(e){ print('Ошибка: $e'); }
}
/*
Что бы ни произошло, я - великолепен!!!
Unhandled exception:
IntegerDivisionByZeroException
*/

```

У разработчиков также имеется возможность перехватывать и обрабатывать, как и в случае с исключениями, ошибки по их типу:

```

try{
    if (inputValue == 0){
        throw ArgumentError();
    }
}on IntegerDivisionByZeroException{
    print('Произошло деление на ноль!!!');
    rethrow;
} on ArgumentError catch(e){
    print('Ошибка: $e'); //Ошибка: Invalid argument(s)
}

```

Иногда бывают ситуации, что необходимо создавать экземпляры абстрактного базового класса с сохранением обязанностей у производных классов, переопределения не реализованных в нем методов. В этом случае абстрактный класс становится обычным и методы класса, которые необходимо реализовать в производном классе, должны генерировать сообщения об ошибках:

```
class Item{
    final String name;
    final double weight;

    Item(this.name, this.weight);
}

class StorageSystem {
    var itemsList = <Item>[];
    final double weightLimit;

    StorageSystem(this.weightLimit);

    void addItem(Item item) => throw NoSuchMethodError;
    Item popItem() => throw NoSuchMethodError;
    double systemWeight()=> throw NoSuchMethodError;
}

class Box extends StorageSystem {
    Box(double weightLimit):super(weightLimit);

    @override
    void addItem(Item item) {
        var currentSystemWeight = systemWeight();
        if((currentSystemWeight+item.weight) < weightLimit){
            itemsList.add(item);
            print('${item.name} добавлен(о/а) в коробку!');
        }
        else{
            print('${item.name} не помещается в коробку!');
        }
    }
}
```

```

@Override
Item popItem() {
    return itemsList.removeLast();
}

@Override
double systemWeight() {
    var sum = 0.0;
    itemsList.forEach((element) {
        sum += element.weight;
    });
    return sum;
}
// методы, характерные для коробки
}

```

```

void main(List<String> arguments) {
    var box = Box(18);
    StorageSystem? storageSystem = box;
    storageSystem.addItem(Item('Книга', 2.6));
    storageSystem.addItem(Item('Чайник', 3.9));
    storageSystem.addItem(Item('Гантеля', 10));
    storageSystem.addItem(Item('Монитор', 4));

    print(storageSystem.popItem().name);
    print(storageSystem.systemWeight());

    StorageSystem? newStorageSystem = StorageSystem(22);
    newStorageSystem.addItem(Item('Монитор', 4));
}
/*
Книга добавлен(о/а) в коробку!
Чайник добавлен(о/а) в коробку!
Гантеля добавлен(о/а) в коробку!
Монитор не помещается в коробку!
Гантеля
6.5
Unhandled exception:
NoSuchMethodError
*/

```

6.3 Пользовательские исключения

Для того, чтобы реализовать ваше собственное исключение и иметь возможность его генерировать в процессе работы приложения, пользовательский класс должен реализовывать интерфейс базового класса для всех исключений – `Exception`:

```
class MyException implements Exception {
  final String? msg;
  const MyException([this.msg]);

  @override
  String toString() => msg ?? 'MyException';
}

void main(List<String> arguments) {
  try{
    throw MyException('Пользовательское исключение');
  } on MyException catch(e){
    print(e); // Пользовательское исключение
  }
}
```

6.4 Трассировка стека

При перехвате исключения с использованием `catch()` для него можно указывать один или два аргумента. Первый аргумент – возникшее исключение, а второй – трассировка стека (объект `StackTrace`):

```
void myFunc(){
  throw MyException('Пользовательское исключение');
}
```

```
void main(List<String> arguments) {
  try{
    myFunc();
  } on MyException catch(e, s){
    print(e);
    print(s);
  }
}
/*
```

Пользовательское исключение

#0 myFunc

bin\my_lib.dart:11

```
#1      main                                bin\my_lib.dart:16
#2      _delayEntrypointInvocation.<anonymous closure> (dart:isolate-
patch/isolate_patch.dart:281:32)
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_
patch.dart:184:12)*//
```

В момент запуска приложения происходит выделение памяти в куче и стеке. Перед самым выделением памяти срабатывают некоторые события и, если в написанном коде существуют ошибки, их можно отследить в стеке. Таким образом, трассировка стека представляет собой список вызовов методов, которые приложение выполняло при возникновении исключения.

6.5 Assert (Утверждение)

Один из способов отладки логики работы приложения в процессе его разработки - assert (утверждения). Они используются для того, чтобы прервать выполнение программы, если утверждение ложно и сообщить о наличии проблемы разработчику. Общая структура утверждения может быть представлена следующим образом:

```
assert(условие, опциональноеСообщение);
```

Для начала посмотрим, как использовать утверждение без прикрепленного к нему опционального сообщения и что при этом выводится, когда условие в утверждении возвращает значение `false`:

```
int myFunc(int a, int b){
  assert(b != 0);
  return a ~/ b;
}

void main(List<String> arguments) {
  print(myFunc(6, 0));
}

//Unhandled exception:
// ... : Failed assertion: line 2 pos 10: 'b != 0': is not true.
```

Теперь добавим сообщение, которое внесет дополнительную ясность в то, с чем связано падение программы:

```
int myFunc(int a, int b){
  assert(b != 0, 'Деление на ноль');
```

```

    return a ~/ b;
}

void main(List<String> arguments) {
  print(myFunc(6, 0));
}
/*
Unhandled exception:
'file:///F:/code/dart/my_lib/bin/my_lib.dart': Failed assertion:
line 2 pos 10: 'b != 0': Деление на ноль
*/

```

Таким образом, первым аргументом `assert` может быть любое выражение, которое возвращает логическое значение (`true` или `false`). Если результат вычисления выражения `true`, утверждение завершается успешно и управление переходит к коду, находящемуся за ним. Если `false`, то утверждение генерирует ошибку `AssertionError`.

Утверждения удобно использовать в процессе разработки программных продуктов. Но необходимо быть внимательным и не помещать в них в качестве выражений различные функции, возвращающие значение логического типа данных, не использовать в утверждениях вызовы методов экземпляров классов и т. д. Это связано с тем, что при `release`-сборке проекта все утверждения в коде игнорируются, то есть они не выполняются.

Резюме по разделу

В данном разделе был рассмотрен механизм исключений, которые представляют собой ошибки, возникающие в ходе выполнения кода и указывающие на имеющиеся проблемы в логике работы программы. Они могут генерироваться автоматически, в процессе выполнения программы или «вручную», самим программистом.

Перехватываемые и обрабатываемые исключения должны специфицироваться, то есть не представлять собой тип базового исключений `Exception`, так как такой подход позволяет точно сказать с ошибкой какого рода в логике работы столкнулось приложение и корректно её обработать. Поэтому, *лучше пусть программа «упадет» и укажет вам на наличие ошибок, чем продолжит работать с некорректными данными.*

Вопросы для самопроверки

1. Что такое исключение? Для чего они используются?
2. Какие два базовых классов для работы с исключениями существуют?
3. Для каких целей используется блок `try`?
4. Для каких целей используется блок `catch`?
5. Для каких целей используется блок `finally`?
6. Какое ключевое слово следует использовать для того, чтобы сгенерировать исключение?
7. Посредством какого ключевого слова можно распространить перехваченное исключение дальше?
8. Что такое трассировка стека? Как использовать данный механизм?
9. Для чего используются утверждения?
10. Всегда ли выполняются утверждения в коде?

Упражнения

1. Напишите класс, проверяющий наличие определенного слова в строке. Если заданное слово не входит в строку – выбрасывается исключение.
2. Напишите класс, в котором перегружены операции деления и умножения. При попытке произвести эти операции со значением ноль, должно выбрасываться исключение.
3. Перехватите исключение из задания 2 и вызовите операцию по новой, передав на вход отличное от нуля значение.
4. Напишите утверждение, которое позволит найти ошибку, если мы обращаемся к элементу списка по не существующему индексу.
5. Реализуйте пользовательское исключение `MySuperException`, которое будет генерироваться всякий раз, если в подаваемом на вход функции списке из целочисленных значений имеется элемент со значением 4.

7 Работа с файлами

В данном разделе рассмотрим, как в Dart осуществляется работа с файлами. Они могут выступать в различном амплуа: от конфигурационных до хранилищ.

Так, например, в конфигурационные файлы выносятся информация, которую может изменять пользователь в настройках приложения и которая влияет на логику его работы. Это делается для большей гибкости, так как хранение такой информации в самом коде программного продукта подразумевает, что при малейшем её изменении необходимо по новой осуществлять сборку (компиляцию) программы.

В некоторых же случаях, в процессе работы приложения необходимо вести логи, чтобы в последствии отследить наличие ошибок или убедиться, что оно работает корректно.

Чаще всего для логирования используются обычные текстовые файлы, а для конфигурации приложений – json-файлы.

7.1 Создание, чтение и запись файла

Для создания, записи, чтения или добавления данных в файл необходимо использовать класс `File`, импортируемый из библиотеки `'dart:io'`. В конструктор создаваемого экземпляра класса `File` передается путь до файла, с которым будет осуществляться работа:

```
import 'dart:io';

void main(List<String> arguments) {
  var myFile = File('text.txt');
}
```

Если в конструкторе указывается только имя файла с его расширением, то подразумевается, что он находится, либо создается в директории проекта. В случае работы с файлом из другой директории, путь до него можно прописать следующим образом:

```
var myFile = File('F:\\code\\text.txt');
```

Если файла по указанному пути не существует, то при попытке прочитать из него данные сгенерируется исключение, а при попытке записать или добавить в него данные, перед началом выполнения

операции произойдет его создание. Сама работа с файлами может производиться как в синхронном, так и в асинхронном режиме. Для работы в синхронном режиме необходимо явно вызывать методы, которые заканчиваются на **Sync**. Дополнительное отличие таких методов заключается в том, что асинхронные всегда возвращают **Future<T>**, в то время как синхронные могут ничего не возвращать (**void**). Более подробное знакомство с асинхронным программированием познакомимся в следующем разделе книги.

Теперь же давайте создадим файл, запишем в него данные и считаем их:

```
import 'dart:io';

void main(List<String> arguments) {
  var myFile = File('text.txt');
  myFile.writeAsStringSync('Привет! О, этот чудный мир!!');
  print(myFile.readAsStringSync()); // чтение всего файла
}
// Привет! О, этот чудный мир!!
```

Метод **writeAsStringSync** имеет следующие аргументы:

```
void writeAsStringSync (
    String contents,
    {FileMode mode = FileMode.write,
    Encoding encoding = utf8,
    bool flush = false}
),
```

где **contents** – записываемые данные, **mode** – текущий режим работы с файлом (по умолчанию - только запись), **encoding** – кодировка записываемых данных (по умолчанию – **utf8**), **flush** – флаг, отвечающий за запись всех буферизированных данных в файл, тем самым производя очистку буфера вывода.

В приведенном выше коде метод **writeAsStringSync** вызывается с параметрами по умолчанию, а это значит, что при повторном запуске произойдет перезапись, а не добавление данных в файл. Чтобы добавить данные в существующий файл, необходимо атрибуту **mode** передать значение **FileMode.append**:

```
import 'dart:io';
```

```
void main(List<String> arguments) {  
  var myFile = File('text.txt');  
  myFile.writeAsStringSync('\nХочу обратно в школу!!!',  
                           mode: FileMode.append);  
  print(myFile.readAsStringSync()); // чтение всего файла  
}  
//Привет! О, этот чудный мир!!  
//Хочу обратно в школу!!!
```

\n отвечает за перевод текста на новую строку, его можно добавлять в любом месте записываемых данных.

Существует несколько режимов работы с файлами:

- **append** – режим открытия файла для чтения и записи в конец. Файл создается, если он еще не существует.
- **read** – режим открытия файла только для чтения.
- **write** – режим открытия файла для чтения и записи. Файл будет перезаписан, если он уже существует. Файл создается, если он еще не существует.
- **writeOnly** – режим открытия файла только для записи. Файл будет перезаписан, если он уже существует. Файл создается, если он еще не существует.
- **writeOnlyAppend** – режим открытия файла только для записи в конец файла. Файл создается, если он еще не существует.

Если необходимо работать с файлом только в определенном режиме (**read**, **write** или **append**), его следует явно указать в методе **open** или **openSync**, после создания экземпляра класса **File**. Также можно использовать методы **openRead** или **openWrite**. Все эти методы возвращают объекты, через которые впоследствии осуществляется работа с файлом в задаваемом режиме:

```
import 'dart:io';  
void main(List<String> arguments) {  
  var myFile = File('text.txt');  
  var sink = myFile.openWrite(mode: FileMode.write);  
  var stringList = <String>['Ий-хо-хо!', 'И', 'бутылка', 'рома!'];  
  sink.writeAll(stringList, ' ');  
}
```

В метод `writeAll` в качестве первого аргумента могут передаваться объекты, поддерживающие итерацию, а второй аргумент представляет собой разделитель между записываемыми данными.

Для более гибкого способа чтения файлов или при работе с файлами большого размера следует использовать поток (`Stream`). Он предоставляет данные в виде блоков байтов и позволяет задавать различные преобразователи, чтобы представить содержимое файла в требуемом формате:

```
import 'dart:io';
import 'dart:convert';
import 'dart:async';

void main() async {
  final myFile = File('text.txt');
  var stringList = <String>[
    'Пятнадцать человек на сундук мертвеца.\n',
    'Йо-хо-хо, и бутылка рому!\n',
    'Тей, и дьявол тебя доведет до конца.\n',
    'Йо-хо-хо, и бутылка рому!'
  ];
  stringList.forEach((element) {
    myFile.writeAsStringSync(element, mode: FileMode.append);
  });

  Stream<String> lines = myFile
    .openRead()
    .transform(utf8.decoder) // Декодирование байтов в UTF-8.
    .transform(LineSplitter()); // Перевод потока в отдельные строки
  try {
    await for (var line in lines) {
      print('$line: ${line.length}');
    }
    print('Файл закрыт');
  } catch (e) {
    print('Ошибка: $e');
  }
}
```

Пятнадцать человек на сундук мертвеца.: 38

```

Йо-хо-хо, и бутылка рому!: 25
Пей, и дьявол тебя доведет до конца.: 36
Йо-хо-хо, и бутылка рому!: 25
Файл закрыт
*/

```

Таким образом, класс File может работать не только с текстовыми представлениями файлов, но и байтовыми.

7.2 Дополнительные методы для работы с файлами

Перед началом работы с файлом имеется возможность проверить - он уже присутствует в системе или нет:

```

import 'dart:io';

void main() async {
  final myFile = File('../text.txt');
  print(myFile.existsSync()); // false
}

```

В данном случае осуществляется проверка существует ли файл `text.txt` на одном уровне выше директории запускаемого приложения.

Также имеется возможность выводить текущее состояние файла, с которым будет осуществляться работа:

```

import 'dart:io';

void main() async {
  final myFile = File('text.txt');
  print(myFile.statSync());
}
/*
FileStat: type file
      changed 2021-04-28 11:29:38.000
      modified 2021-04-28 13:23:21.000
      accessed 2021-04-28 15:49:33.000
      mode rw-rw-rw-
      size 224
*/

```

Текущую директорию запускаемого приложения, от которой потом уже отталкиваться для построения пути к файлу, можно узнать следующим способом:

```
import 'dart:io';

void main() async {
  final rootPath = Directory.current.path;
  print(rootPath); // F:\code\dart\my_lib
}
```

Для удаления файла используйте метод `deleteSync` или `delete`:

```
import 'dart:io';

void main() async {
  final myFile = File('text.txt');
  print(myFile.existsSync()); // true
  myFile.deleteSync();
  print(myFile.existsSync()); // false
}
```

Дополнительно имеется возможность явно создавать файлы, без записи в них какого-либо значения. Для этого существуют методы `createSync` и `create`, на вход которых атрибуту `recursive` подается булево значение (по умолчанию `false`). Если данный параметр был установлен в `true`, при наличии отсутствующих каталогов в пути к файлу, они будут созданы, после чего создастся сам файл. В противном случае файл создается только тогда, когда все каталоги на его пути уже существуют. Если таковых не имеется – сгенерируется исключение `FileSystemException`:

```
import 'dart:io';

void main() async {
  final myFile = File('data/text.txt');
  print(myFile.existsSync()); // false
  myFile.createSync(recursive: true);
  print(myFile.existsSync()); // true
}
```

Для получения размера файла в байтах используйте метод `length` и `lengthSync`:

```
import 'dart:io';

void main() async {
  final myFile = File('data/text.txt');
  myFile.writeAsStringSync('Мама мыла раму');
  print(myFile.lengthSync()); // 26
  myFile.writeAsStringSync('\nПотом батарею...');
  print(myFile.lengthSync()); // 29
}
```

Время последней модификации файла можно узнать посредством методов `lastModifiedSync` и `lastModified`:

```
import 'dart:io';

void main() async {
  final myFile = File('data/text.txt');
  print(myFile.lastModifiedSync()); // 2021-04-28 17:24:49.000
}
```

Для более подробного ознакомления с методами класса `File` обратитесь к документации Dart.

7.3 Работа с JSON-файлами

7.3.1 Что за зверь такой JSON?

JSON (JavaScript Object Notation) - довольно простой и повсеместно используемый формат обмена данными. Его достаточно удобно читать. Это относится как к человеку, так и к компьютеру. Дополнительным достоинством JSON является то, что он по сути представляет собой текстовый формат данных и полностью независим от языка реализации. То есть сохраненные данные в JSON-формате средствами одного языка программирования могут быть использованы в приложении, которое реализовано на другом языке. Именно поэтому он обрел огромную популярность и широкое распространение.

В основе JSON находятся 2 структуры:

- Коллекция пар «ключ:значение» (объект). На разных языках программирования концепция такого *объекта* реализована по-своему. В Dart, в качестве такой коллекции, используется тип данных `Map<K, V>`;

- Упорядоченный список значений (массив, список, последовательность и т. д.).

Таким образом, каждый объект представляет из себя неупорядоченную коллекцию пар «ключ:значение», чье тело заключается в фигурные скобки: {**объект**}. Ключом может выступать только строковый тип данных. Каждый ключ заключается в двойные кавычки ("**имяКлюча**"), после которых идет разделительный символ «:» (двоеточие), а пары «ключ:значение» разделяются между собой запятой.

Значение может представлять собой: строку в двойных кавычках, число, логический тип данных (**true/false**), **null**, объект или массив. То есть объект может содержать в себе другой объект, массив из объектов или прочих свойств.

В качестве примера давайте приведем описание объекта **Студент** в JSON-формате, у которого есть имя, возраст, флаг состояния холост или нет, номер курса обучения и его краткая характеристика:

```
{
  "name": "Alex",
  "age": 35,
  "course": 2,
  "single": true,
  "description": [
    "Мечтатель",
    "Ленив",
    "Студент",
    "Постоянно жалуется на жизнь"
  ]
}
```

А теперь представим, что нам необходимо описать целую группу студентов:

```
{
  "groupName": "2-МД-66",
  "course": 2,
  "amountStudents": 25,
  "students":[
    {
      "name": "Alex",
      "age": 35,
      "course": 2,
      "single": true,
      "description": [
        "Мечтатель",
```

```

        "Ленив",
        "Студент",
        "Постоянно жалуется на жизнь"
    ]
},
{
    "name": "Max",
    "age": 15,
    "course": 2,
    "single": true,
    "description": [
        "Реалист",
        "Мега-мозг",
        "Студент"
    ]
},
...,
{
    "name": "Smith",
    "age": 20,
    "course": 2,
    "single": false,
    "description": [
        "Паникер",
        "Двоечник",
        "Студент",
        "Девиз: жизнь - боль..."
    ]
}
]
}

```

7.3.2 Сохранение данных в JSON-формате

Объявим класс Student с некоторыми параметрами и реализуем возможность их сохранения в json-файл:

```

import 'dart:convert';
import 'dart:io';

class Student {
    final String name;
    int age;
    int course;
    bool single;
    List<String> _descriptionList = [];

```

```

Student(
    {required this.name,
     required this.age,
     required this.course,
     required this.single});

void addDescription(String description) {
    _descriptionList.add(description);
}

void addAllDescriptions(List<String> descriptions) {
    _descriptionList.addAll(descriptions);
}

@override
String toString() {
    var student = 'Студент {имя: $name, возраст: $age, ';
    student += 'курс: $course, холост: $single, ';
    student += 'описание: $_descriptionList}';
    return student;
}

Map<String, dynamic> toJson() {
    return <String, dynamic>{
        'name': name,
        'age': age,
        'course': course,
        'single': single,
        'description': _descriptionList
    };
}

}

void main() async {
    var myFile = File('data/student.json');
    var student = Student(name: 'Alex', age: 19, course: 1, single: false);
    var descriptions = ['Мечтатель', 'Ленив', 'Студент'];
    student.addAllDescriptions(descriptions);
    student.addDescription('Постоянно жалуется на жизнь');
    print(student);
}

```

```
// Код ниже автоматически вызывает метод toJson у
// передаваемого на вход экземпляра класса Student
var encoder = JsonEncoder.withIndent(' ');
myFile.writeAsStringSync(encoder.convert(student));
print(myFile.readAsStringSync());
}
/*
  Студент {имя: Alex, возраст: 19, курс: 1, холост: false, описание:
[Mечтатель, Ленив, Студент, Постоянно жалуется на жизнь]}
{
  "name": "Alex",
  "age": 19,
  "course": 1,
  "single": false,
  "description": [
    "Мечтатель",
    "Ленив",
    "Студент",
    "Постоянно жалуется на жизнь"
  ]
}
*/
```

7.3.3 Загрузка данных из JSON-файла

Теперь перепишем код таким образом, чтобы создаваемый экземпляр класса Student мог инициализироваться данными, загружаемыми из записанного ранее json-файла:

```
import 'dart:convert';
import 'dart:io';

class Student {
  final String name;
  late int age;
  late int course;
  late bool single;
  List<String> _descriptionList = [];

  Student(
    {required this.name,
    required this.age,
    required this.course,
    required this.single});
```

```

Student.fromJson(Map<String, dynamic> json)
: name = json['name'] {
  age = json['age'];
  course = json['course'];
  single = json['single'];
  _descriptionList = List<String>.from(json['description']);
}

void addDescription(String description) {
  _descriptionList.add(description);
}

void addAllDescriptions(List<String> descriptions) {
  _descriptionList.addAll(descriptions);
}

@override
String toString() {
  var student = 'Студент {имя: $name, возраст: $age, ';
  student += 'курс: $course, холост: $single, ';
  student += 'описание: $_descriptionList}';
  return student;
}

Map<String, dynamic> toJson() {
  return <String, dynamic>{
    'name': name,
    'age': age,
    'course': course,
    'single': single,
    'description': _descriptionList
  };
}

void main() async {
  var myFile = File('data/student.json');
  var student = Student.fromJson(jsonDecode(myFile.readAsStringSync()));
  print(student);
}

```

```
/*
```

```
  Студент {имя: Alex, возраст: 19, курс: 1, холост: false,  
  описание: [Мечтатель, Ленив, Студент, Постоянно жалуется на  
  жизнь]}
```

```
*/
```

7.3.4 Библиотека `json_serializable` для десериализации и сериализации объектов в JSON-формат

Под сериализацией понимается процесс преобразования какой-либо структуры данных или параметров экземпляра класса в формат (последовательность байт), позволяющий осуществлять хранение и передачу этих данных. Десериализация – это обратный процесс, то есть из полученных, загруженных или прочитанных данных мы восстанавливаем структуру данных, объект или экземпляр класса с необходимыми параметрами.

Показанный ранее способ для сериализации и десериализации данных экземпляров класса `Student` хорош только для маленьких проектов, а когда речь идет о среднем или огромном количестве передаваемых данных и наличии вложенных объектов, то самостоятельно прописывать под это дело код – то еще удовольствие. В таких случаях на помощь приходят библиотеки, позволяющие автоматически генерировать код сериализации и десериализации классов. От разработчика требуется только правильно осуществить аннотацию классов, для которых этот код будет сгенерирован.

В Dart для генерации кода сериализации и десериализации пользовательских классов рекомендуется использовать библиотеку `json_serializable`. Для её подключения откройте файл `pubspec.yaml` и внесите в него следующие изменения:

```
environment:  
  sdk: '>=2.12.0 <3.0.0'  
  
dependencies:  
  json_serializable: ^4.1.1
```

Если попробуем сразу писать код с использованием аннотаций этой библиотеки, то он не соберется. Это связано с тем, что данный функционал классов еще не был сгенерирован. Для решения этой проблемы внесем еще одно дополнение в файл `pubspec.yaml`:

```
dev_dependencies:  
  pedantic: ^1.10.0  
  build_runner: ^2.0.1
```

Дополнительно в переменных средах (переменная Path) необходимо будет прописать явный путь до директории <bin> SDK Flutter, после чего перезагрузить VS Code:

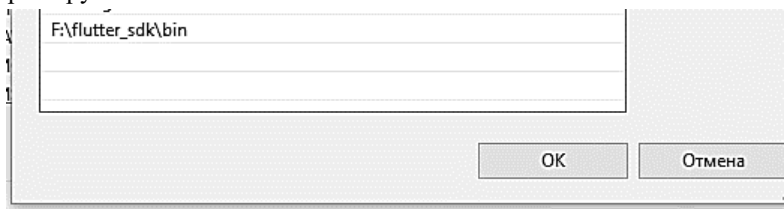


Рисунок 7.1 – Дополнительная настройка окружения

Для запуска **build_runner** можно использовать следующие команды:

- **build**: запускает отдельную сборку и завершает работу;
- **watch**: запускает постоянный сервер сборки, который следит за файловой системой на предмет изменений и при необходимости выполняет перестройку;
- **serve**: то же самое, что и **watch**, но также запускает сервер разработки. По умолчанию он обслуживает веб-каталог и тестовый каталог на портах 8080 и 8081;
- **test**: запускает отдельную сборку, создает объединенный выходной каталог, а затем запускает команду `pub run test --precompiled <merged-output-dir>`.

Теперь напишем с использованием аннотаций код класса *Group* и *Student*:

```
// student.dart  
import 'package:json_annotation/json_annotation.dart';  
  
part 'student.g.dart'; // сюда будет сгенерирован  
                        // код сериализации/десериализации  
  
@JsonSerializable()  
class Student {  
  final String name;
```

```

late int age;
late int course;
late bool single;
@JsonKey(name: 'description')
List<String>? _descriptionList = [];

Student(this.name, this.age, this.course,
        this.single, List<String>? description){
    _descriptionList = description;
}

Student.withOutDescription(
    {required this.name,
    required this.age,
    required this.course,
    required this.single});

factory Student.fromJson(Map<String, dynamic> json) =>
    _$StudentFromJson(json);
Map<String, dynamic> toJson() => _$StudentToJson(this);

List<String> get description{
    return _descriptionList ?? <String>[];
}

void addDescription(String description) {
    _descriptionList?.add(description);
}

void addAllDescriptions(List<String> descriptions) {
    _descriptionList?.addAll(descriptions);
}

@override
String toString() {
    var student = 'Студент {имя: $name, возраст: $age, ';
    student += 'курс: $course, холост: $single, ';
    student += 'описание: $_descriptionList}';
    return student;
}
}

```



```

//group.dart
import 'package:json_annotation/json_annotation.dart';

import 'student.dart';

part 'group.g.dart';

@JsonSerializable()
class Group{
  final String groupName;
  late int course;
  List<Student>? students = [];

  Group({required this.groupName,
        required this.course, this.students});

  Group.emptyGroup({required this.groupName,
                    required this.course});

  factory Group.fromJson(Map<String, dynamic> json) =>
    _$GroupFromJson(json);
  Map<String, dynamic> toJson() => _$GroupToJson(this);

  int get amountStudents{
    return students?.length ?? 0;
  }

  void addStudent(Student student){
    students?.add(student);
  }

  void addAllStudents(List<Student> students){
    this.students?.addAll(students);
  }

  @override
  String toString() {
    var groupInfo = 'Группа: $groupName \nкурс: $course\n';
    groupInfo += 'кол-во студентов: $amountStudents \nсписок: [ \n';
    students?.forEach((element) {
      groupInfo += '$element \n';
    });
  }
}

```

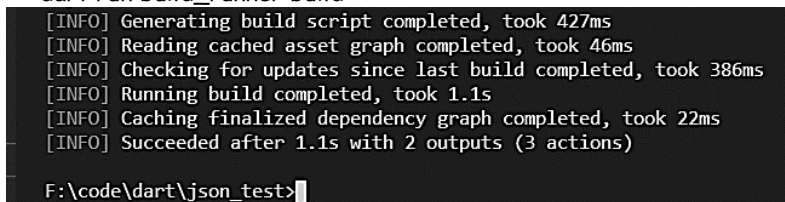
```

});
groupInfo += ']';
return groupInfo;
}
}

```

Не переживайте, что часть кода подсвечивается, это нормально. Чтобы избавиться от этого навязчивого красного подчеркивания запустим в терминале проекта в VS Code следующую команду:

```
dart run build_runner build
```



```

[INFO] Generating build script completed, took 427ms
[INFO] Reading cached asset graph completed, took 46ms
[INFO] Checking for updates since last build completed, took 386ms
[INFO] Running build completed, took 1.1s
[INFO] Caching finalized dependency graph completed, took 22ms
[INFO] Succeeded after 1.1s with 2 outputs (3 actions)

F:\code\dart\json_test>

```

Рисунок 7.2 – Запуск build_runner

После успешной генерации кода в директории появятся 2 новых файла: `student.g.dart` и `group.g.dart`, которые будут содержать автоматически сгенерированный код:

```
// student.g.dart
```

```
part of 'student.dart';
```

```
// *****
```

```
// JsonSerializableGenerator
```

```
// *****
```

```

Student _$StudentFromJson(Map<String, dynamic> json) {
  return Student(
    json['name'] as String,
    json['age'] as int,
    json['course'] as int,
    json['single'] as bool,
    (json['description'] as List<dynamic>?)?.map((e) => e as String).toList(),
  );
}

```

```

Map<String, dynamic> _$StudentToJson(Student instance) => <String,
dynamic>{

```

```

        'name': instance.name,
        'age': instance.age,
        'course': instance.course,
        'single': instance.single,
        'description': instance.description,
    };

// group.g.dart
part of 'group.dart';

// *****
// JsonSerializableGenerator
// *****

Group _$GroupFromJson(Map<String, dynamic> json) {
    return Group(
        groupName: json['groupName'] as String,
        course: json['course'] as int,
        students: (json['students'] as List<dynamic>?)
            ?.map((e) => Student.fromJson(e as Map<String, dynamic>))
            .toList(),
    );
}

Map<String, dynamic> _$GroupToJson(Group instance) => <String, dynamic>{
    'groupName': instance.groupName,
    'course': instance.course,
    'students': instance.students,
};

```

Осталось написать немного кода для проверки корректности работы сериализации и десериализации:

```

//main.dart
import 'dart:convert';
import 'dart:io';

import 'student.dart';
import 'group.dart';

void main(List<String> arguments) {
    // формируем фгруппу из нескольких студентов

```

```

var studentsGroup = Group.emptyGroup(groupName: '1-МД-66',
    course: 1);
var firstStudent = Student.withOutDescription(name: 'Alex',
    age: 19, course: 1, single: false);
var descriptions = ['Мечтатель', 'Ленив', 'Студент'];
firstStudent.addAllDescriptions(descriptions);
firstStudent.addDescription('Постоянно жалуется на жизнь');
var secondStudent = Student('Maxim', 22, 1, true, descriptions);
secondStudent.addDescription('Девиз: всё нормально!');
studentsGroup.addAllStudents(<Student>[firstStudent,
    secondStudent]);

// сериализуем группу в json
var encoder = JsonEncoder.withIndent(' ');
var test = encoder.convert(studentsGroup);

var myFile = File('data/group.json');
myFile.createSync(recursive: true);
myFile.writeAsStringSync(test); // записываем в файл

// десериализуем данные и выводим в терминал
var newStudentsGroup = Group.fromJson(jsonDecode(test));
print(newStudentsGroup);
}
/*
Группа: 1-МД-66
курс: 1
кол-во студентов: 2
список: [
  Студент {имя: Alex, возраст: 19, курс: 1, холост: false, описание:
[Mечтатель, Ленив, Студент, Постоянно жалуется на жизнь]}
  Студент {имя: Maxim, возраст: 22, курс: 1, холост: true, описание:
[Mечтатель, Ленив, Студент, Девиз: всё нормально!]}
]
*/

```

Записанный файл group.json будет иметь вид:

```

{
  "groupName": "1-МД-66",
  "course": 1,
  "students": [
    {

```

```

        "name": "Alex",
        "age": 19,
        "course": 1,
        "single": false,
        "description": [
            "Мечтатель",
            "Ленив",
            "Студент",
            "Постоянно жалуется на жизнь"
        ]
    },
    {
        "name": "Maxim",
        "age": 22,
        "course": 1,
        "single": true,
        "description": [
            "Мечтатель",
            "Ленив",
            "Студент",
            "Девиз: всё нормально!"
        ]
    }
]
}
}

```

Следует особо внимательно подходить к наличию приватных переменных в классе при использовании библиотеки `json_serializable`. Так, например, в классе `Student` список описания студента является приватным и его имя для сериализации экземпляра класса в json-формат переопределено через аннотацию `@JsonKey(name: 'description')`. При этом имеется геттер с таким же названием и в конструкторе для передачи списка описания студента используется аргумент с именем `description`. Библиотека `json_serializable` при генерации кода делает акцент на то, чтобы имена объявленных переменных совпадали с именами аргументов конструктора и не начинались с нижнего подчеркивания. Это связано с тем, что генерация кода основывается на конструкторе сериализуемого и десериализуемого класса. Если какая-то переменная не будет присутствовать в качестве его аргумента, код для неё не будет сгенерирован.

Обратите внимание на сохраненный файл, там нет информации о том, сколько студентов учится в группе. Конечно, их там сейчас не там

много, и мы можем посчитать это в уме. А если их там будет 20 или 30? Сгенерированный код можно модифицировать, но обычно это не рекомендуется делать, так как в случае какого-либо изменения в файле, по которому этот код генерировался при повторном запуске команды `dart pub run build_runner build` все внесенные изменения затрутсся. А теперь представьте, что у вас таких мест много и во все вы внесли изменения модифицировав код. Чем такой подход будет лучше написания сериализации и десериализации вручную? Ничем!!! Можно попросту забыть об этих изменениях, а потом при очередной пересборке проекта удивляться почему от так сейчас работает или не работает вовсе.

Поэтому, если хотите, чтобы такой параметр, как число студентов в группе сериализовался в json-формат, то придется его использовать в основном конструкторе класса. В виду этого используйте основной конструктор для генерации кода перевода в json-формат и из него, а экземпляры класса в системе создавайте через именованные конструкторы.

Резюме по разделу

В данном разделе мы рассмотрели почти все основные возможности Dart по работе с файлами. Почему почти все? Потому, что Dart еще может манипулировать данными в html-файлах, но с появлением Flutter for Web эти его возможности теряют свою актуальность. Да и давайте признаемся честно, не так уж и много компаний горит желанием использовать его в этих целях, тогда как JavaScript предоставляет куда больший набор библиотек и возможностей.

Также был рассмотрен такой формат представления данных, как json. Он часто будет встречаться на вашем пути и чем раньше получится с ним подружиться, тем лучше.

Использование библиотек для генерации кода довольно удобно, но не спешите их использовать везде, так как они тянут за собой ряд зависимостей в ваш проект, с которыми в последствии придется считаться. Особенно, если одну из них прекратит поддерживать её автор, да и сообществу до неё не будет никакого дела.

Вопросы для самопроверки

1. Какой класс в Dart позволяет работать с файлами?

2. Какие режимы работы с файлами существуют? В чем их различия?
3. Какой класс лучше использовать при необходимости прочитать/загрузить файл большого размера?
4. Как проверить существование файла в системе?
5. Каким образом можно получить путь до директории запускаемого приложения?
6. Как явным образом создать файл? За что отвечает флаг `recursive`?
7. Что такое JSON (JavaScript Object Notation)? Для чего и где он используется?
8. Что такое сериализация и десериализация?
9. Для чего можно использовать библиотеку `json_serializable`? Какие у неё ограничения?
10. Стоит ли всегда использовать библиотеки для генерации кода? Почему?

Упражнения

1. Напишите реализацию шкафа, который может хранить различные вещи. Реализуйте функционал для записи его текущего состояния в json-файл без использования библиотеки `json_serializable`.
2. Реализуйте функционал из задания 1 с помощью библиотеки `json_serializable`.
3. Напишите класс машина, у которой имеются следующие состояния: стоп, поворот налево, поворот направо, движение вперед, разгон. Каждый раз, когда у неё меняется состояние, должна осуществляться запись в текстовый логирующий файл. Формат записи в лог-файл: 1) дата и время изменения состояния; 2) предыдущее состояние; 3) текущее состояние.
4. Напишите две реализации логгера, которые наследуются от одного базового класса. Первая реализация будет осуществлять запись в текстовый файл, вторая в json-файл. Используйте подход работы через интерфейс базового класса и добавьте возможность в предыдущую задачу менять реализацию логгера, записывающего изменение состояния машины.
5. Измените код класса `Group`, рассмотренного в данной теме, таким образом, чтобы в json-файл дополнительно сохранялась информация о количестве студентов в группе.

8 Асинхронное программирование и Isolate

Dart – однопоточный язык программирования, но это совсем не значит, что у вас нет возможности писать код, который будет выполняться параллельно. Для этих случаев используются **Isolate**. Если сравнивать концепцию **Isolate** с инструментами для параллельного программирования в других языках, то ближе всего будет такое понятие, как процесс. Это связано с тем, что каждый **Isolate** работает со своей областью памяти и может обмениваться с другими **Isolat**-ами данными посредством сообщений. Основной поток выполнения программы на Dart также представляет собой **Isolate**.

Весь код на Dart выполняется последовательно, то есть за раз выполняется одна операция. Таким образом, если в основном коде приложения вызывать выполнение функции, то управление перейдет к ней и придется дожидаться пока она не вернет результат (выполнит возлагаемую на неё работу). А если в функции будет выполняться довольно большой объем вычислений, то с точки зрения пользователя, ваша программа «зависнет». Такое поведение приложения указывает на неправильное использование процессорного времени. Что уж говорить, о негодовании пользователя, когда практически в каждом компьютере установлены многоядерные процессоры.

Асинхронное программирование позволяет отложить процесс выполнение функции, не останавливая выполнение основного кода и задать функцию, которая обработает возвращаемый ей результат. То есть асинхронная функция будет выполнена немного позже, в освободившееся процессорное время. Но необходимо понимать следующее: **выполнится она всё в том же основном потоке приложения**. Поэтому не следует в функциях, объявляемых как асинхронные, производить сложные вычисления, а тем более прописывать вечные циклы.

Перед тем как перейдем к изучению инструментов асинхронного программирования в Dart рассмотрим, что такое цикл событий (event loop) [14]. Это позволит писать более качественный асинхронный код, в котором вас будет поджидать куда меньшее количество сюрпризов, чем в том случае, когда не имеете никакого представления о Event Loop.

8.1 Event Loop архитектура в Dart

8.1.1 Базовая концепция цикла и очереди событий

В каждом приложении с графическим пользовательским интерфейсом (GUI) реализована концепция цикла событий и очереди событий. Именно они гарантируют, что любые графические операции и события (движение или щелчки мышкой, нажатие клавиш и т. д.) обрабатываются по очереди. То есть каждый попадающий в очередь событий элемент берется оттуда циклом событий, после чего обрабатывается и так до тех пор, пока в очереди есть элементы, которые представляют собой операции ввода/вывода, таймеры и т. д. Рассмотрим очередь событий, которая содержит события таймера и ввода данных пользователем:

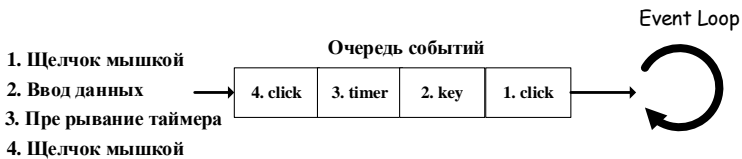


Рисунок 8.1 – Концепция обработки события из очереди циклом событий

В рамках Dart, концепция обработки очереди событий из рис. 8.1, может быть представлена следующим образом:

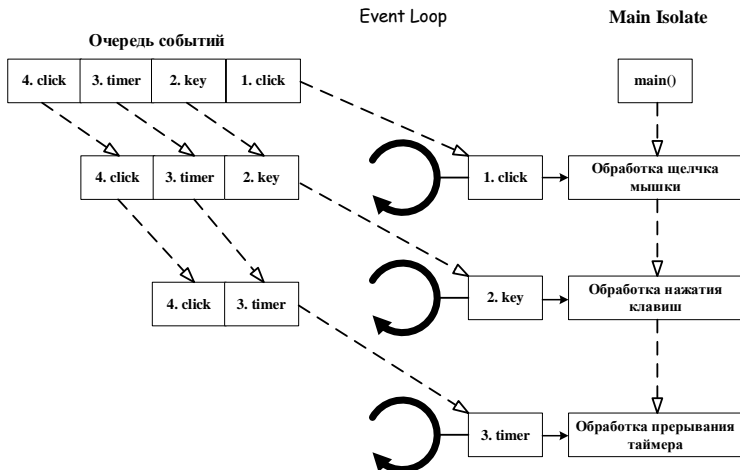


Рисунок 8.2 – Пример обработки очереди циклом событий в Dart

Из рис. 8.2 видно, что все элементы очереди событий обрабатываются в главном потоке приложения.

8.1.2 Очереди и цикл событий в Dart

Каждое Dart-приложение имеет один цикл событий, который осуществляет работу с двумя очередями:

- очередь событий. Содержит как события Dart, так и все внешние события: ввод/вывод, таймеры, сообщения между экземплярами `Isolate` и т. д.;
- очередь микрозадач. Используется для очень коротких внутренних действий, которые необходимо выполнять асинхронно, сразу после завершения какого-либо события и перед передачей управления обратно в очередь событий.

В качестве примера элемента для помещения в очередь микрозадач может выступать задача удаления ресурса (файла и т. д.), после того как он был закрыт. Так как этот процесс может занимать какое-то время, то его разумнее всего выполнить в асинхронном режиме.

Цикл обработки событий начинает свою работу при выходе потока управления из функции верхнего уровня `main`. Первым делом он выполняет любые микрозадачи в порядке их нахождения в очереди микрозадач. Следом за этим начинается обработка первого элемента в очереди событий, то есть он извлекается из очереди и обрабатывается. Затем идет повторение цикла: сначала выполняются все микрозадачи, а после обрабатывается следующий элемент в очереди событий. Когда обе очереди пусты и больше не ожидается событий, приложение закрывается.

Ниже представлена структурная схема алгоритма работы цикла событий:

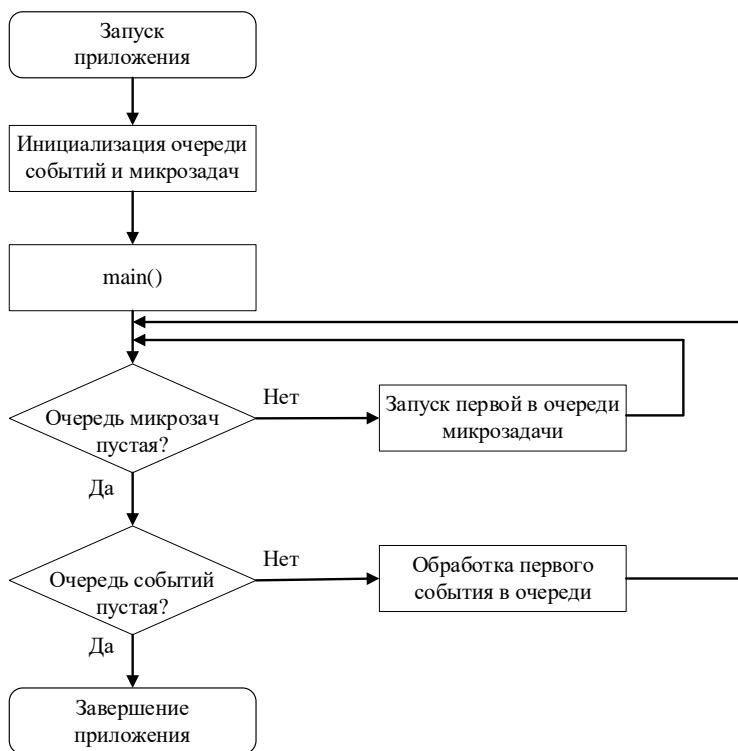


Рисунок 8.2 – Алгоритм работы цикла событий

Посмотрите внимательно структурную схему алгоритма работы цикла событий. Из неё следует, что пока цикл событий выполняет задачи из очереди микрозадач, обработка элементов очереди событий не производится. То есть приложение не может рисовать графику, обрабатывать события ввода/вывода и т. д.

И хотя теперь, лучше разбираясь в алгоритме работы цикла событий, имеется возможность предсказать порядок выполнения задач, нельзя точно сказать, когда цикл событий будет доставать на обработку задачу из очереди. Это связано с тем, что сама система обработки событий Dart основана на однопоточном цикле. Таким образом, при создании очередной отложенной задачи, событие ставится в очередь, но оно не может быть обработано до тех пор, пока не будут обработаны все события, находящиеся перед ним в очереди.

Для добавления очередного элемента в очередь событий, код которого должен выполняться позже, в Dart используется класс **Future**.

А для добавления нового элемента в конец очереди микрозадач используется функция верхнего уровня `scheduleMicrotask`, либо именованный конструктор `Future.microtask`.

Обычно рекомендуется при планировании отложенных задач использовать класс `Future`, тем самым помещая их в очередь событий. Это помогает сохранить короткую очередь микрозадач, тем самым уменьшая вероятность того, что из-за неё будет простаивать очередь событий. В тех же случаях, когда задача обязательно должна завершиться до того, как будут обработаны какие-либо элементы из очереди событий, немедленно вызывайте выполнение функции для её обработки. Если этого сделать не получается, помещайте в очередь микрозадач.

В качестве примера работы алгоритма цикла событий реализуем программу, выводящую в терминал строки. Сначала поместим событие вывода строки в очередь событий, а потом в очередь задач:

```
import 'dart:async';

void main(List<String> arguments) {
  Future(() => print('1-й элемент очереди событий'));
  Future(() => print('2-й элемент очереди событий'));

  Future.microtask((){
    print('1-й элемент очереди микрозадач');
  });
  scheduleMicrotask((){
    print('2-й элемент очереди микрозадач');
  });
}
/*
1-й элемент очереди микрозадач
2-й элемент очереди микрозадач
1-й элемент очереди событий
2-й элемент очереди событий
*/
```

8.2 Асинхронное программирование

Для того, чтобы ваш код имел возможность выполняться асинхронно, в заголовке модуля необходимо импортировать библиотеку `dart:async`. После этого вам станут доступны такие классы, как `Future` и

Stream. В Dart также имеются ключевые слова `async` и `await`, которые позволяют писать асинхронный код, внешне мало чем отличающийся от синхронного.

К наиболее частым задачам, где код должен выполняться асинхронно можно отнести:

- получение данных по сети;
- запись в базу данных;
- чтение данных из файла.

8.2.1 Future API

Future API позволяет добавлять задачи как в очередь событий, так и в очередь микрозадач для их отложенного асинхронного выполнения. Каждая задача может завершиться успешно, либо в процессе её работы сгенерируется исключение, которое следует обработать или дать распространиться дальше, вплоть до падения приложения.

В Dart существует такое понятие, как **future** (фьючерс/будущее). Под ним понимается объект, представляющий собой результат вычисления, которое, возможно, еще не произошло и его результат может быть известен когда-нибудь в будущем. Говоря простыми словами, **future** – экземпляр класса `Future<T>`, позволяющий нам писать асинхронный код и предоставляющий доступ к результату вычисления, где `T` – тип возвращаемого результата.

Экземпляр класса **Future** может быть создан с использованием одного и следующих конструкторов:

- `Future(FutureOr<T> computation())` создает **future**, содержащий результат асинхронного вызова `computation` с помощью `Timer.run`.
- `Future.delayed(Duration duration, [FutureOr<T> computation()])` создает **future**, вычисление которого выполняется после задержки, указываемой в `duration`. Необязательный аргумент конструктора `computation` представляет собой ссылку на функцию, которая будет выполняться после задаваемой задержки.
- `Future.error(Object error, [StackTrace? stackTrace])` создает **future**, который будет завершен ошибкой `error`. Это дает достаточно времени для добавления обработчика ошибки. В том случае, если обработчик не будет добавлен до завершения **future**, ошибка будет считаться необработанной.

- `Future.microtask(FutureOr<T> computation())` создает `future`, который посредством функции `scheduleMicrotask` помещает функцию `computation` в очередь микрозадач (во всех остальных случаях идет работа с очередью событий), запускает её асинхронно и возвращает результат.

- `Future.sync(FutureOr<T> computation())` создает `future`, содержащий результат немедленного вызова `computation`.

- `Future.value([FutureOr<T> value])` создает `future`, содержащее значение `value`.

Почти во всех приведенных конструкторах класса `Future` тип возвращаемого значения передаваемой функции в качестве аргумента `computation` указывается как `FutureOr<T>`. Это значит, что передаваемая функция должна возвращать либо `Future<T>` либо объект типа `T`:

```
import 'dart:async';

int add() => 10 + 15;

void main(List<String> arguments) {
  Future<int> future = Future(add);
}
```

Чтобы получить вычисляемое значение функции `add` у `future` необходимо вызвать метод `then`, куда передать callback-функцию (функция обратного вызова) с типом входного аргумента, соответствующему типу возвращаемого значения. Callback-функция будет вызвана сразу, как обработается элемент очереди событий, соответствующий `future`:

```
import 'dart:async';

int add() => 10 + 15;

void main(List<String> arguments) {
  var firstFuture = Future<int>(add);
  Future(()=> print('Oo'));
  firstFuture.then((value) => print(value));
  print('завершение main');
}
/*
```

завершение main

25

Oo

***/**

Также, вывести в терминал предыдущие значения можно следующим способом:

```
import 'dart:async';
```

```
int add() => 10 + 15;
```

```
void myPrint(int a) => print(a);
```

```
void main(List<String> arguments) {  
  var firstFuture = Future<int>(add);  
  var secondFuture = Future<int>(() => 'Oo');  
  firstFuture.then(myPrint);  
  secondFuture.then(print);  
  print('завершение main');  
}
```

/*

завершение main

25

Oo

***/**

Callback-функция может возвращать объекты различного типа, тем самым организуя цепочки из методов **then**, которые будут вызываться друг за другом:

```
import 'dart:async';
```

```
void main(List<String> arguments) {  
  var future = Future<String>(() =>  
    'Привет! Это событие в очереди под номером: ');  
  
  var newfuture = Future<String>(() =>  
    'Привет! Это еще одно событие в очереди под номером: ');  
  
  int a = 10;  
  future.then((value){  
    print('$value 1');  
    return 1;  
  });  
}
```

```

    }).then((value) => print(value + a));

    newfuture.then((value){
      print('$value 2');
      return 2.5;
    }).then((value) => print(value + a));

    print('завершение main');
  }
  /*
завершение main
Привет! Это событие в очереди под номером: 1
11
Привет! Это еще одно событие в очереди под номером: 2
12.5
*/

```

Если в процессе асинхронного выполнения задачи сгенерируется исключение, то оно вернется как результат выполнения отложенной операции. `Future` позволяет перехватывать все возникающие в процессе отложенного выполнения исключения и ошибки посредством метода `catchError`. При отсутствии обработки исключений, либо невозможности обработать исключение сгенерированного типа, оно будет распространено дальше, что в итоге приведет к завершению программы. Так, например, в коде ниже будет производиться обработка всех возможных исключений:

```

import 'dart:async';

class MyException implements Exception {
  final String? msg;

  const MyException([this.msg]);

  @override
  String toString() => msg ?? 'MyException';
}

int myFunction(){
  var sum = 0;
  for(var i=0; i<30; i++){

```



```

        sum += i;
        if(sum > 40){
            throw MyException();
        }
    }
    return sum;
}

void main(List<String> arguments) {
    var future = Future<int>(myFunction);
    future.then(print)
        .catchError((onError) => print(onError));
    Future(()=> print('_-'));
    print('завершение main');
}
/*
завершение main
MyException
_-
*/

```

Теперь в метод `catchError` добавим дополнительную проверку, чтобы он осуществлял обработку исключений или ошибок только заданного типа. Для этого используется его второй именованный (необязательный) аргумент `test`, которому передается функция, принимающая на вход экземпляр исключения и возвращающая логическое значение. Если возвращается `true`, то значит перехватили нужное исключение и будем его обрабатывать в методе `catchError`, иначе исключение распространится дальше.

```

void main(List<String> arguments) {
    var future = Future<int>(myFunction);
    future.then(print)
        .catchError((onError) => print(onError),
            test: (error) => error is MyException);
    Future(()=> print('_-'));
    print('завершение main');
}

/*
завершение main

```

MyException

```
--  
*/
```

В этом случае исключение обработалось, и программа завершилась в штатном режиме, но если в функции `myFunction` будет генерироваться отличное от `MyException` исключение (например, просто `Exception`), то её выполнение завершится намного раньше и в терминале будут красоваться следующие строки:

```
завершение main  
Unhandled exception:  
Exception
```

Когда же присутствует острая необходимость при успешном выполнении фьючерса или наличии ошибки выполнить какое-либо действие (например, закрыть доступ к ресурсу и т. д.), на помощь приходит метод `whenComplete`:

```
void main(List<String> arguments) {  
    var future = Future<int>.delayed(  
        Duration(seconds: 3),  
        myFunction  
    );  
    future.then(print)  
        .catchError((onError) => print(onError),  
            test: (error) => error is MyException)  
        .whenComplete(() => print('Я всё равно лучший!!!!'));  
    Future(() => print('--'));  
    print('завершение main');  
}  
/*  
завершение main  
--  
Я всё равно лучший!!!!  
Unhandled exception:  
Exception  
*/
```

8.2.2 Ключевые слова `async` и `await`

Для упрощения написания функций, которые должны выполняться асинхронно, существуют ключевые слова `async` и `await`. Эти два слова довольно тесно связаны друг с другом, поскольку ключевое слово `await` можно использовать только в теле тех функций, которые помечены, как `async`. К тому же тип возвращаемого результата функции должен быть обернут в `Future`:

```
String getBigData() {  
    return 'Гигатонны информации');  
}  
  
Future<void> makeRequestData() async {  
    print('Запрос данных');  
    var data = await getBigData();  
    print(data);  
    print('Данные получены');  
}  
  
void main(List<String> arguments) {  
    print('Запуск main');  
    makeRequestData();  
    print('Завершение main');  
}  
/*  
Запуск main  
Запрос данных  
Завершение main  
Гигатонны информации))  
Данные получены  
*/
```

Код в асинхронной функции `makeRequestData` выполняется синхронно (просто примите это ^_^) вплоть до первого вызова `await`, которое представляет собой асинхронную операцию. То есть она не блокирует выполнение других операций, позволяя им выполняться до своего завершения. Таким образом, когда поток управления в функции `makeRequestData` встречается с ключевым словом `await`, он останавливается до тех пор, пока вызываемая функция `getBigData` не

вернет свой результат. После чего продолжается выполнение функции `makeRequestData`.

В результате выполнения `await` всегда возвращается объект `Future`. А в том случае, когда вызываемая функция возвращает значение отличного от `Future` типа данных, Dart автоматически оборачивает его в `Future`. При этом, возвращаемое значение вызываемой через `await` функции может быть получено путем обычного присваивания.

Если в функции `getBigData` сгенерируется исключение, его следует обработать посредством конструкции `try...catch...finally`:

```
String getBigData() {  
    throw Exception('Прервалось соединение!!!');  
}
```

```
Future<void> makeRequestData() async {  
    print('Запрос данных');  
    try {  
        print(await getBigData());  
        print('Данные получены');  
    } catch (e) {  
        print('Что-то пошло не так: $e');  
    }  
}
```

```
void main() {  
    print('Запуск main');  
    makeRequestData();  
    print('Завершение main');  
}  
/*
```

Запуск main

Запрос данных

Что-то пошло не так: Exception: Прервалось соединение!!!

Завершение main

***/**

Количество вызовов функций с использованием ключевого слова `await` в теле функции, помеченной как `async`, не ограничено. Необходимо помнить только то, что вызываться они будут последовательно:

```
String getBigData() {
```

```

    return 'Гигатонны информации));'
}

String changeData(String data) {
    return data.toUpperCase();
}

Future<void> makeRequestData() async {
    print('Запрос данных');
    var data = await getBigData();
    print(data);
    print('Данные получены');
    print('Пристааем к изменению данных');
    print(await changeData(data));
    print('Данные изменены');
}

void main(List<String> arguments) {
    print('Запуск main');
    makeRequestData();
    print('Завершение main');
}
/*
Запуск main
Запрос данных
Завершение main
Гигатонны информации))
Данные получены
Пристааем к изменению данных
ГИГАТОННЫ ИНФОРМАЦИИ))
Данные изменены
*/

```

8.2.3 Stream (Поток)

Поток (Stream) в Dart – это последовательность асинхронных событий, которые подразделяются на три типа:

- событие данных (элемент потока);
- событие ошибки (что-то пошло не так);
- событие "done", оповещающее всех слушателей (тех, кто подписался на поток) о его завершении.

Основное преимущество от использования потоков заключается в том, что код остается слабосвязанным. Это связано с тем, что классу, в котором создается экземпляр потока, отвечающий за выдачу готовых данных, не нужно ничего знать о том, кто подписался (слушает) на получение событий и почему. Аналогичная ситуация обстоит и с потребителями данных. Они должны только придерживаться интерфейса потока, в то время как источник данных от них полностью скрыт.

Для управления потоками в Dart [17] используются следующие классы:

- **Stream**. Представляет асинхронный поток данных. Слушатели могут подписаться на получение уведомлений о появлении новых событий данных.
- **EventSink**. Обратный поток, добавление событий данных в который направляет эти данные в подключенный поток.
- **StreamController**. Упрощает управление потоками, автоматически создавая поток и приемник, а также предоставляя методы для управления поведением потока.
- **StreamSubscription**. Экземпляры этого класса могут сохранять ссылку на подписку, что позволяет им приостанавливать, возобновлять или отменять поток данных.

В большинстве случаев нет необходимости напрямую создавать экземпляры классов **Stream** и **EventSink**. Это связано с тем, что при создании экземпляра класса **StreamController**, автоматически создается поток и приемник.

Примером потока может выступать изменение положения курсора мыши, список простых чисел, получаемые по сети данные и т. д. На каждый поток имеется возможность подписаться (прослушать поток), путем задания одной или нескольких callback-функций, которые будут вызываться при добавлении в него новых данных. Самый простой пример использования потоков – написание асинхронной генераторной функции:

```
Stream<int> myGenerator(int last) async* {  
  for (var i = 0; i <= last; i++) {  
    yield i;  
  }  
}
```

```
void createGenerator(int lastValue) async {
```

```

var stream = myGenerator(lastValue);
// слушаем поток и выводим получаемые данные в терминал
stream.listen((s) => print(s));
}

void main(List<String> arguments) {
  print('Запуск main');
  createGenerator(20);
  print('Завершение main');
}
/*
Запуск main
Завершение main
0
...
20
*/

```

Также потоки предоставляют возможность асинхронно итерироваться по существующим последовательностям. Для этого используется именованный конструктор `Stream.fromIterable`:

```

void iterableStream(List<int> list) {
  var stream = Stream.fromIterable(list);
  print('Начало работы потока');
  stream.listen(
    (s) => print(s),
  );
  print('Завершение работы потока');
}

void main(List<String> arguments) {
  iterableStream([1, 2, 3, 4, 5]);
}
/
* Начало работы потока
Завершение работы потока
1
...
5
*/

```

Обратите внимание на вывод в терминал в предыдущем примере. Итерация по списку происходила в асинхронном режиме. Порой может возникнуть ситуация, что основной код функции, обрабатывающей события потока, должен выполнять в синхронном режиме. Для этого следует использовать конструкцию `await for`:

```
void iterableStream(List<int> list) async {  
    var stream = Stream.fromIterable(list);  
    print('Начало работы потока');  
    await for (var num in stream) {  
        print(num);  
    }  
    print('Завершение работы потока');  
}
```

```
void main(List<String> arguments) {  
    iterableStream([1, 5]);  
}  
/*
```

```
Начало работы потока  
1  
5  
Завершение работы потока  
*/
```

При использовании `await for` для обработки исключения следует обрабатывать посредством конструкции `try...catch...finally`. В том случае, когда функция обработки задается через метод `listen` экземпляра класса `Stream`, для обработки исключений следует использовать его именованный аргумент `onError`:

```
Stream<int> myGenerator(int last) async* {  
    for (var i = 0; i <= last; i++) {  
        if (i >= 2) {  
            throw Exception('Ошибка!!!');  
        }  
        yield i;  
    }  
}
```

```
void createGenerator(int lastValue) async {  
    var stream = myGenerator(lastValue);
```



```

// слушаем поток и выводим получаемые данные в терминал
stream.listen((s) => print(s),
              onError: (e) => print(e);
)

void main(List<String> arguments) {
  print('Запуск main');
  createGenerator(20);
  print('Завершение main');
}
/*
Запуск main
Завершение main
0
1
Exception: Ошибка!!!
*/

```

Теперь давайте разберемся, как осуществляется работа с экземпляром класса `StreamController`:

```

import 'dart:async';

void main(List<String> arguments) {
  final controller = StreamController<String>();

  final subscription = controller.stream.listen((String data) {
    print(data);
  });

  controller.add('Привет!!!');
  controller.add('И еще раз, Привет!!!');
}
// Привет!!!
// И еще раз, Привет!!!

```

Экземпляр класса `StreamController` предоставляет доступ к потоку для прослушивания и реагирования на события. Для этого используется метод `listen` экземпляра класса `Stream`, посредством которого задается функция обратного вызова, обрабатывающая поступающие данные в поток посредством метода `add`. Сам же метод потока `listen` возвращает

экземпляр `StreamSubscription`, позволяющий управлять подпиской на поток.

Давайте представим, что у нас имеется кофемашина состоящая из монетоприемника и блока приготовления кофе. Блок приготовления подписывается на события поступления денег в монетоприемник и после того, как накапливается пороговая сумма, начинается приготовление капучино:

```
import 'dart:async';

class CoinAcceptor{
  final _addCoin = StreamController<int>();
  Stream<int> get dataStream => _addCoin.stream;

  void addCoin(int coin) => _addCoin.add(coin);
}

class CoffeMachine{
  int valueCoins = 0;

  CoffeMachine(Stream<int> stream){
    stream.listen(addCoin);
  }

  void addCoin(int coin){
    valueCoins += coin;
    if(valueCoins >=30){
      print('Готовим капучино!');
    }
    print('Общее кол-во монет: $valueCoins');
  }
}

void main(List<String> arguments) {
  print('Запуск main');
  var coinAcceptor = CoinAcceptor();
  var coffeMachine = CoffeMachine(coinAcceptor.dataStream);
  coinAcceptor.addCoin(25);
  coinAcceptor.addCoin(4);
  coinAcceptor.addCoin(3);
  print('Завершение main');
}
```

```

/*
Запуск main
Завершение main
Общее кол-во монет: 25
Общее кол-во монет: 29
Готовим капучино!
Общее кол-во монет: 32
*/

```

В случае необходимости класс `CoffeMachine` можно написать более компактным образом:

```

class CoffeMachine{
    int valueCoins = 0;

    CoffeMachine(Stream<int> stream){
        stream.listen((coin){
            valueCoins += coin;
            if(valueCoins >=30){
                print('Готовим капучино!');
            }
            print('Общее кол-во монет: $valueCoins');
        });
    }
}

```

В приведенном примере работы кофемашины у монетоприемника может быть только один подписчик на события. Когда же имеется необходимость разрешить несколько слушателей потока, следует создать широковебательный поток посредством именованного конструктора `Stream<T>.broadcast`:

```

final _addCoin = StreamController<int>.broadcast ();

```

Посредством потоков имеется возможность передавать не только значения встроенных типов данных, но и пользовательских:

```

import 'dart:async';

class Coin{
    final int value;
    Coin(this.value);
}

```

```

class CoinAcceptor{
  final _addCoin = StreamController<Coin>();
  Stream<Coin> get dataStream => _addCoin.stream;

  void addCoin(Coin coin) => _addCoin.add(coin);
}

class CoffeMachine{
  int valueCoins = 0;

  CoffeMachine(Stream<Coin> stream){
    stream.listen((coin){
      valueCoins += coin.value;
      if(valueCoins >= 30){
        print('Готовим капучино!');
      }
      print('Общее кол-во монет: $valueCoins');
    });
  }
}

void main(List<String> arguments) {
  print('Запуск main');
  var coinAcceptor = CoinAcceptor();
  var coffeMachine = CoffeMachine(coinAcceptor.dataStream);
  coinAcceptor.addCoin(Coin(35));
  print('Завершение main');
}
/*
Запуск main
Завершение main
Готовим капучино!
Общее кол-во монет: 35
*/

```

Для более исчерпывающей информацией по работе с потоками и способам управления подписками обратитесь к документации Dart.

8.3 Isolate (Изоляты)

Несмотря на то, что Dart запускает весь свой код в одном изоляте, при необходимости имеется возможность создавать пользовательские изоляты, имеющие свою собственную память и единственный поток выполнения, который запускает цикл обработки событий. То есть каждый новый изолят получает свой собственный цикл событий и свою собственную память, к которой другие изоляты не имеют доступа.

Единственный способ, благодаря которому изоляты могут работать вместе – обмен сообщениями. Так один изолят может отправить сообщение другому, который обработает полученное сообщение, используя свой цикл событий.

Несмотря на имеющиеся у такого подхода недостатки, у него также есть ряд преимуществ [19]:

- Выделение памяти и сборка мусора в изолированном объекте не требуют блокировки.
- Есть только один поток и если он не занят, то память не изменяется.

Давайте реализуем эхо-изолят и разберем принцип работы кода приведенного примера:

```
import 'dart:isolate';
```

```
class IsolatesMessage<T> {  
  final SendPort sender;  
  final T message;  
  
  IsolatesMessage({  
    required this.sender,  
    required this.message,  
  });  
}
```

```
late SendPort isolateSendPort;  
late Isolate isolate;
```

```
Future<void> createIsolate() async {  
  var receivePort = ReceivePort();  
  isolate = await Isolate.spawn(  
    echoCallbackFunction,  
    receivePort.sendPort,  
  );  
}
```

```

    isolateSendPort = await receivePort.first;
}

Future<String> sendReceive(String send) async{
    var port = ReceivePort();
    isolateSendPort.send(
        IsolatesMessage<String>(
            sender: port.sendPort,
            message: send,
        )
    );
    return await port.first;
}

void echoCallbackFunction(SendPort sendPort){
    var receivePort = ReceivePort();
    // возвращаем ссылку на порт для отправки данных
    // в изолят
    sendPort.send(receivePort.sendPort);
    receivePort.listen((message) {
        // обработчик принимаемых сообщений изолятом
        var isolateMessage = message as IsolatesMessage<String>;
        print('Isolate: ${isolateMessage.message}');
        isolateMessage.sender.send(isolateMessage.message);
    });
}

void main()async{
    await createIsolate();
    print('Main: ${await sendReceive('Старт!')}');
    print('Main: ${await sendReceive('1')}');
    print('Main: ${await sendReceive('2')}');
    print('Main: ${await sendReceive('3')}');
}
/*
Isolate: Старт!
Main: Старт!
Isolate: 1
Main: 1
Isolate: 2
Main: 2
Isolate: 3

```

Для создания экземпляра изолята используется статический метод `Isolate.spawn<T>`, в нем первым аргументом выступает ссылка на функцию, которая будет выполняться в изоляте и принимает в качестве входного аргумента сообщение типа данных `T`, а вторым – ссылка на экземпляр сообщения, которое поступает в изолят сразу при его создании. В нашем случае это ссылка на порт, через который вернется ссылка на порт самого изолята для последующего обмена сообщениями с ним. Класс `IsolatesMessage` используется для работы с изолятом. В его первом параметре передается ссылка на порт для последующей передачи из изолята сообщения главному приложению. `ReceivePort` должен создаваться каждый раз при обращении к изоляту, поэтому метод `createIsolate` отвечает за создание изолята и его инициализацию, то есть получения от него ссылки порт, через который в последующем будет осуществляться пересылка сообщения в изолят. Посредством метода `sendReceive` осуществляется вся остальная работа с созданным изолятом: передается сообщение из основного приложения, ожидается ответ, после чего он возвращается в основное приложение.

Когда экземпляр класса `Isolate` становится более не нужным, его работу рекомендуется завершить посредством метода `kill`, который принимает на вход один из следующих параметров:

- `Isolate.immediate`. Изолят завершает свою работу как можно скорее. Так как управляющие сообщения обрабатываются по порядку, то все ранее отправленные управляющие события из этого изолята будут обработаны. Завершение работы изолята должно произойти не позднее, чем при вызове метода с параметром `Isolate.beforeNextEvent`. То есть оно может произойти раньше, если у системы есть способ полностью завершить работу в более раннее время, даже во время выполнения другого события.
- `Isolate.beforeNextEvent` (используется по умолчанию). Завершение работы планируется до следующего возврата управления в цикл событий принимающего изолятора.

Если по логике работы не подразумевается длительная работа с изолятом и он создается на один раз, то вместо ссылки на порт для

пересылки сообщений в изолят, возвращайте сам результат его выполнения.

8.4 Что и когда использовать?

Вот несколько советов, которые помогут вам определиться какие из рассматриваемых в данной теме механизмов и когда использовать:

- Если части кода не должны быть прерваны, используйте обычный синхронный процесс (один метод или несколько методов, которые вызывают друг друга);
- Если фрагменты кода могут работать независимо, без влияния на плавность работы приложения (отсутствие зависаний), используйте **Future**;
- Если работа может занять некоторое время и потенциально вызывать задержки в работе графического пользовательского интерфейса приложения, используйте **Isolate**.

Также при выборе того, использовать **Future** или **Isolate** можно ориентироваться на среднее время, необходимое для выполнения кода:

- **Future**, если выполнение метода занимает пару миллисекунд.
- **Isolate**, если время работы метода может занимать несколько сотен и более миллисекунд.

Резюме по разделу

В текущем разделе мы с вами рассмотрели возможности асинхронного и параллельного программирования в Dart, что такое цикл событий (**Event Loop**) и принципы его работы.

Future API, а также ключевые слова **async** и **await** предоставляют довольно большой набор возможностей при написании асинхронного кода, задачи по выполнению которого помещаются в очередь событий и при необходимости могут быть добавлены в очередь микрозадач. Первым делом цикл событий выполняет любые микрозадачи в порядке их нахождения в очереди микрозадач. После того, как очередь микрозадач становится пустой, начинается обработка первого элемента в очереди событий. Затем идет повторение этого цикла. Когда же обе очереди становятся пустыми и больше не ожидается событий, приложение закрывается.

Isolate позволяет писать параллельный код и так как весь код, который запускается в изолятах работает со своей областью памяти, то для обмена данными между ними используются сообщения. Даже основной поток выполнения программы на Dart запускается в собственном изоляте.

Если выполнения задачи не занимает больше нескольких микросекунд – используйте асинхронное программирование, в противном случае задумайтесь о возможности использования изолятов.

Вопросы для самопроверки

1. Что такое **Event Loop** и каковы принципы его работы в Dart?
2. В чем отличие очереди микрозадач от очереди событий?
3. Для чего используется очередь событий?
4. В какой последовательности в терминал выведутся сообщения из следующего кода:

```
void main() async{
  print("А");
  await Future((){
    print("Б");
    Future(()=>print("В"));
    Future.microtask(()=>print("Г"));
    Future(()=>print("Д"));
    print("Е");
  });
  print("Ж");
}
```

5. Когда при использовании **Future API** задачи добавляются в очередь событий или микрозадач?
6. Для чего используется метод **then** экземпляра класса **Future**?
7. Как обрабатываются исключения при использовании **Future API** и ключевых слов **async** и **await**? В чем разница?
8. Что такое **Stream** (Поток) в Dart и для чего он используется?
9. Какие классы используются для управления потоками в Dart? В чем их отличия?
10. Что такое **Isolate**? Опишите принципы работы с изолятами.
11. Опишите случаи, когда лучше использовать асинхронное программирование.

12. Опишите случаи, когда лучше использовать изоляты.

Упражнения

1. Напишите функцию для асинхронной записи данных в файл.
2. Напишите функцию для асинхронного чтения данных из файла.
3. Напишите код для записи данных в json-файл (например, картотеки книг) в изоляте.
4. Напишите код для чтения данных из json-файла (например, картотеки книг) в изоляте. Верните сформированный список экземпляров класса Книга в основной поток приложения.
5. Напишите код для расчета числа Фибоначчи в асинхронной функции.
6. Напишите код для расчета числа Фибоначчи в изоляте.

9 Сетевое программирование

В настоящее время трудно представить приложение, которое не осуществляет работу по сети. Очень часто нам приходится взаимодействовать с различными сервисами для отправки им или получения данных, их последующей обработки и визуализации. Либо самим организовывать клиент-серверную архитектуру у разрабатываемого программного продукта.

Dart предоставляет довольно обширные возможности для организации такой работы посредством библиотеки `dart:io`. Вследствие чего довольно просто написать собственный TCP, UDP или HTTP сервер, а также клиентские приложения для работы с ними.

9.1 Передача данных между сервером и клиентом по протоколу TCP

Под протоколом обычно понимают некоторый своеобразный язык, используемый по договоренности двумя сторонами (компьютерными программами) для общения между собой. Все межсетевое взаимодействие осуществляется посредством интерфейса сокетов, которые могут реализовывать различные транспортные механизмы.

Для создания TCP-сервера используется класс `ServerSocket` и его статический метод `bind`, в который передается IP и номер порта сервера (хоста), после чего можно задать анонимную функцию для асинхронной обработки каждого нового соединения к серверу.

При написании клиентской части используется класс `Socket` и его статический метод `connect`, который создает новое соединение с задаваемым хостом и портом и возвращает `Future`, завершающийся одним из перечисленных образов:

- После подключения к серверу возвращается экземпляр класса `Socket`.
- Из-за того, что не удалось найти заданный хост и порт, генерируется исключение.

Для следующего примера понадобится два проекта. Один для серверной части, а другой для клиентской. Так как реализовывать простое клиент-серверное эхо-приложение не очень-то и интересно, давайте напомним код, в котором на стороне клиента формируется

экземпляр класса `Student`, после чего он сериализуется в json-формат и передается на сервер, где выполняется его десериализация и отправляется приветственное сообщение клиенту. После приема сообщения как клиентом, так и сервером будем вызывать функцию `exit`, иначе они будут продолжать свою работу.

Так как клиентская и серверная часть представляют собой два разных проекта, в каждый из них необходимо поместить класс `Student`, либо выделить его в отдельную библиотеку и подключить к каждому из проектов. Мы же пойдем по самому легкому пути:

```
//server.dart
import 'dart:convert';
import 'dart:io';

class Student {
  final String name;
  late int age;
  late int course;
  late bool single;
  List<String> _descriptionList = [];

  Student(
    {required this.name,
     required this.age,
     required this.course,
     required this.single});

  Student.fromJson(Map<String, dynamic> json)
    : name = json['name'] {
    age = json['age'];
    course = json['course'];
    single = json['single'];
    _descriptionList = List<String>.from(json['description']);
  }

  void addDescription(String description) {
    _descriptionList.add(description);
  }

  void addAllDescriptions(List<String> descriptions) {
    _descriptionList.addAll(descriptions);
  }
}
```

```

}

@override
String toString() {
    var student = 'Студент {имя: $name, возраст: $age, '
    student += 'курс: $course, холост: $single, '
    student += 'описание: $_descriptionList}';
    return student;
}

Map<String, dynamic> toJson() {
    return <String, dynamic>{
        'name': name,
        'age': age,
        'course': course,
        'single': single,
        'description': _descriptionList
    };
}

}

void main() {
    print('Запуск main');
    ServerSocket.bind('127.0.0.1', 8084)
    .then((serverSocket) {
        serverSocket.listen((socket) {
            // обрабатываем соединение очередного клиента
            // с сервером
            socket.cast<List<int>>>()
            .transform(utf8.decoder).listen((event) {
                // обрабатываем данные, которые поступают
                // от клиента
                print(event);
                var student = Student.fromJson(jsonDecode(event));
                print(student);
                // отправка сообщения клиенту
                socket.write('Hello, Client');
                exit(0); // завершение работы приложения
            });
        });
    });
}

```

```
    print('Завершение main');
}
```

//client.dart

```
import 'dart:convert';
import 'dart:io';
```

```
class Student {
  //Без изменений относительно server.dart
}
```

```
void main() {
  print('Запуск main');
  startTCPClient();
  print('Завершение main');
}
```

```
void startTCPClient() async {
  var student = Student(name: 'Alex', age: 19, course: 1, single: false);
  var descriptions = ['Мечтатель', 'Ленив', 'Студент'];
  student.addAllDescriptions(descriptions);
  student.addDescription('Постоянно жалуется на жизнь');
  var encoder = JsonEncoder.withIndent(' ');

  // соединяемся с сервером
  var socket = await Socket.connect('127.0.0.1', 8084);
  socket.write(encoder.convert(student));
  socket.cast<List<int>>().transform(utf8.decoder).listen((event) {
    print(event);
    exit(0);
  });
}
```

В процессе работы клиентской части в терминал будет выведено:

```
Запуск main
Завершение main
Hello, Client
```

А на стороне сервера:

```
Запуск main
Завершение main
```

```
{
  "name": "Alex",
  "age": 19,
  "course": 1,
  "single": false,
  "description": [
    "Мечтатель",
    "Ленив",
    "Студент",
    "Постоянно жалуется на жизнь"
  ]
}
```

Студент {имя: Alex, возраст: 19, курс: 1, холост: false, описание: [Мечтатель, Ленив, Студент, Постоянно жалуется на жизнь]}

9.2 Передача данных между сервером и клиентом по протоколу UDP

Код организации соединения, что на стороне сервера, что на стороне клиента будет не очень-то и отличаться. В обоих случаях используется класс `RawDatagramSocket` с его статическим методом `bind`, который осуществляет связывание с задаваемым хостом и портом и возвращает `Future<RawDatagramSocket>`:

```
//server.dart
import 'dart:convert';
import 'dart:io';

class Student {
  //Без изменений относительно server.dart
}

void main() {
  print('Запуск main');
  startUDPServer();
  print('Завершение main');
}

// UDP
void startUDPServer() async {
  var rawDgramSocket =
    await RawDatagramSocket.bind(InternetAddress.loopbackIPv4, 8083);
```

```

await for (RawSocketEvent event in rawDgramSocket) {
  if (event == RawSocketEvent.read) {
    var datagram = rawDgramSocket.receive();
    var json = utf8.decode(datagram!.data);
    print(json);
    var student = Student.fromJson(jsonDecode(json));
    print(student);
    rawDgramSocket.send(
      utf8.encode('Hello, Client'), InternetAddress.loopbackIPv4, 8084);
    rawDgramSocket.close();
    exit(0);
  }
}
}

```

//client.dart

```

import 'dart:convert';
import 'dart:io';

```

```

class Student {
  //Без изменений относительно server.dart
}

```

```

void main() {
  print('Запуск main');
  startUDPClient();
  print('Завершение main');
}

```

```

void startUDPClient() async {
  var student = Student(name: 'Alex', age: 19, course: 1, single: false);
  var descriptions = ['Мечтатель', 'Ленив', 'Студент'];
  student.addAllDescriptions(descriptions);
  student.addDescription('Постоянно жалуется на жизнь');
  var encoder = JsonEncoder.withIndent(' ');

  var rawDgramSocket = await RawDatagramSocket.bind('127.0.0.1', 8084);
  rawDgramSocket.send(utf8.encode(encoder.convert(student)),
    InternetAddress('127.0.0.1'), 8083);
}

```



```

await for (RawSocketEvent event in rawDgramSocket) {
  if (event == RawSocketEvent.read) {
    var datagram = rawDgramSocket.receive();
    print(utf8.decode(datagram!.data));
    rawDgramSocket.close();
    exit(0);
  }
}
}
}

```

Вывод в терминал сообщений, что со стороны сервера, что со стороны клиента будет идентичен предыдущему примеру с TCP. Обратите внимание на номера портов. Обе стороны (клиент и сервер) должны знать IP и номер порта принимающей стороны. В противном случае не получится организовать передачу данных по сети между клиентом и сервером.

Как можно заметить из двух рассмотренных примеров, существуют различные способы работы с сокетами: используя `await for` или метод `then`, вызываемый сразу после метода `bind`.

9.3 HTTP-сервер

Для написания функционала, который позволит получать данные с использованием протокола HTTP через REST API используется класс `HttpServer`. Он представляет собой поток, предоставляющий объекты `HttpRequest`, где каждый такой объект связан с объектом `HttpResponse`. Таким образом, в ходе обработки запроса, сервер передает ответы на сторону клиента, используя объект `HttpResponse` экземпляра класса обрабатываемого запроса.

Давайте напишем такой HTTP-сервер, при обращении к которому можно будет получить json-представление студента или приветствие от сервера:

```

import 'dart:convert';
import 'dart:io';

class Student {
  //Без изменений
}

```

```

void main() {
    print('Запуск main');
    startHTTPServer();
    print('Завершение main');
}

void startHTTPServer() {
    var student = Student(name: 'Alex', age: 19, course: 1, single: false);
    var descriptions = ['Мечтатель', 'Ленив', 'Студент'];
    student.addAllDescriptions(descriptions);
    student.addDescription('Постоянно жалуется на жизнь');
    var encoder = JsonEncoder.withIndent(' ');
    var answer = encoder.convert(student);

    HttpServer.bind(InternetAddress.loopbackIPv4, 8080).then((server) {
        server.listen((HttpRequest request) {
            print(request.uri.path);
            if (request.uri.path.startsWith('/student')) {
                request.response.write(answer);
                request.response.close();
            } else if (request.uri.path.startsWith('/hello')) {
                request.response.write('Добро пожаловать на сервер!');
                request.response.close();
            } else {
                request.response.write('Дратути!');
                request.response.close();
            }
        });
    });
}

```

Теперь откройте браузер и в поле указания пути до ресурса напишите строку `http://127.0.0.1:8080/student` :

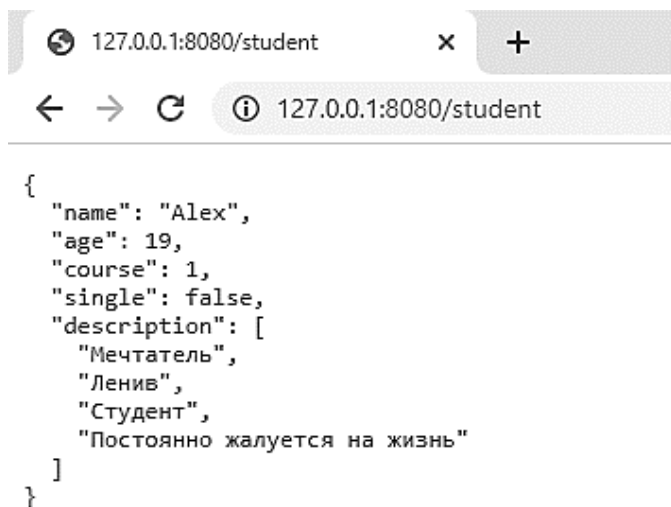


Рисунок 9.1 – Ответ HTTP-сервера на запрос

Если в строке запроса **student** изменить на **hello**, то сервер ответит следующим образом:

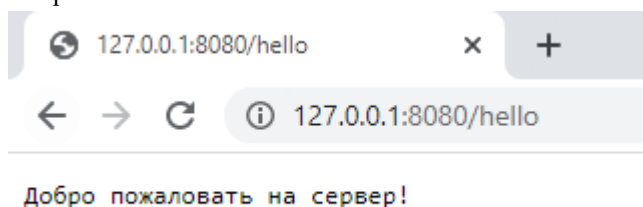


Рисунок 9.2 – Ответ HTTP-сервера на запрос

А при попытке обращения к серверу по запросу, который он не может обработать, всегда будет выводиться следующий ответ:

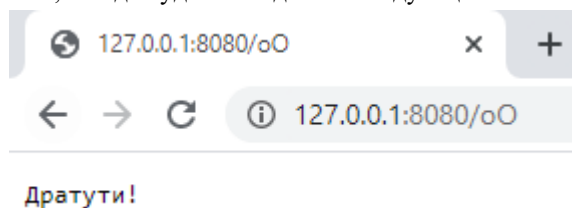


Рисунок 9.3 – Ответ HTTP-сервера на необрабатываемый запрос

Так как при обращении к серверу, в соответствии со спецификацией протокола HTTP, запросы могут быть сформированы с использованием

метода GET, POST и т. д., чтобы узнать о том с каким представлением метода пришел запрос на сервер, у него необходимо обратиться к атрибуту `method (request.method)`.

Для более полного представления о возможностях Dart для разработки back-end составляющей, обратитесь к документации [20].

Резюме по разделу

В данном разделе мы рассмотрели базовые механизмы Dart для организации межсетевого взаимодействия. Как разрабатывать собственные клиент-серверные приложения с использованием протокола TCP и UDP, а также как написать HTTP-сервер.

При организации межсетевого взаимодействия не забывайте закрывать сокет, если он больше не нужен и корректно выходить из приложения. Так как при экстренной остановке приложения могут оставаться открытые соединения, из-за чего не получится его перезапустить с исходными параметрами портов.

Вопросы для самопроверки

1. Какая библиотека Dart используется для написания приложений, поддерживающих работу по сети?
2. Какой класс необходимо использовать при написании TCP-сервера?
3. Какой класс необходимо использовать при написании TCP-клиента?
4. Какой класс необходимо использовать при написании клиента и сервера, работающих по протоколу UDP?
5. В чем отличие реализации приложения с клиент-серверной архитектурой, использующей протокол TCP от UDP?
6. Какой класс необходимо использовать при написании HTTP-сервера? Что он из себя представляет?
7. Как организовать передачу данных в формате json от HTTP-сервера клиенту?

Упражнения

Для следующих упражнений придется погрузиться в недра поисковика, так как они сформулированы таким образом, что материал по части из них рассматривался либо не в полной мере, либо не

рассматривался вовсе. Это сделано специально! Не забывайте о том, что один из навыков, которым вы должны обладать – поиск информации в интернете и её корректное применение в своем проекте. Не следует довольствоваться просто найденным ответом и бездумно копировать найденный код. Лучше перепечатайте его вручную и более подробно разберитесь в том, как работает один из вариантов (или все варианты) ответов.

1. Напишите приложение, которое способно передавать файлы с сервера клиенту по протоколу TCP.
2. Напишите приложение, которое способно передавать файлы с сервера клиенту по протоколу UDP.
3. Напишите приложение (клиентскую и серверную часть), для общения пользователей внутри локальной сети с использованием протокола TCP.
4. Напишите приложение (клиентскую и серверную часть), для общения пользователей внутри локальной сети с использованием протокола UDP.
5. Напишите сетевой калькулятор, где на клиентской стороне пользователь вводит выражение, а само его вычисление производится на серверной части, после чего ответ возвращается клиенту.
6. Напишите приложение (клиентскую и серверную часть) для перевода значений в различную систему счисления (десятичная, восьмеричная, двоичная, шестнадцатеричная). С клиента передаются следующие данные: значение, его текущая система счисления, в какую систему счисления необходимо перевести. После того, как на сервере осуществится перевод значения, верните его клиенту.

10 Тестирование

Можно привести много доводов за и против тестирования, но необходимо понимать одно – покрытие проекта тестами уже многие годы де-факто является стандартом в IT-индустрии. К сожалению, начинающие разработчики рассматривают тестирование так достаточно нудный и трудоемкий процесс, забирающий время от их любимого дела – программирования. Но одно дело, если вы пишете проект «для себя», а совсем другое, когда им будет пользоваться большое количество людей. Поэтому выпускать на рынок продукт, не покрытый тестами – себе дороже.

К тому же существует методология разработки программного обеспечения, где тестирование является основной составляющей – Test-Driven Development (разработка через тестирование). Ключевая идея TDD – написание кода теста перед самим процессом кодирования какого-либо класса разрабатываемого программного продукта. Это позволяет довольно хорошо покрыть код тестами и полностью автоматизировать сам процесс тестирования, запуская его при каждом внесении изменений программистом в реализацию того или иного компонента. Дополнительным плюсом от применения TDD является упрощение программной архитектуры, то есть, когда модуль, компонент или класс проходит написанный под него тест, он считается готовым. При обычном же подходе к разработке велика вероятность, что написанный код получится избыточным, из-за того, что программист может пуститься «во все тяжкие», аргументируя это фразой: «А что, если?».

Существует довольно большое количество видов тестирования, но в документации Dart акцент делается только на трех [21]:

- **Модульное тестирование.** Такие тесты сосредоточены на проверке мельчайших частей тестируемого программного обеспечения: функции, методы или классы.
- **Компонентное тестирование.** Данные тесты необходимы для проверки того, что компонент (обычно состоит из нескольких классов) ведет себя должным образом. При компонентном тестировании часто требует использование фиктивных объектов, которые могут имитировать: действия пользователя, события и т. д.
- **Интеграционное и сквозное тестирование.** Эти тесты проверяют поведение всего приложения или его большей части.

Интеграционный тест зачастую выполняется на реальном устройстве, симуляторе операционной системы или в браузере. Он состоит из двух частей: самого приложения и тестового приложения для его проверки.

Для написания тестов в Dart принято использовать пакет (библиотеку) **test**, а когда необходимо использовать объекты заглушки (фиктивные объекты) – **mockito**.

10.1 Установка пакета **test** в проект

Для установки пакета **test** в текущий проект пропишите его в качестве зависимости проекта в файле **pubspec.yaml**:

```
dev_dependencies:  
  test: ^1.17.3 # актуальная версия на момент написания книги
```

Обычно, при создании консольного проекта, этот пакет автоматически прописывается как зависимость, но, если вы создаете простой консольный проект это придется сделать вручную. Также в основной директории проекта придется создать каталог «**test**», где будут находиться файлы тестового окружения проекта.

10.2 Написание тестов

Название каждого файла с тестом должно заканчиваться постфиксом **_test**. Перед тем, как приступить к написанию самих тестов, в директории **lib** текущего проекта создадим файл **my_functions.dart** в котором напомним следующий код:

```
import 'dart:math';  
  
int add(int a, int b) => a + b;  
int sub(int a, int b) => a - b;  
int mul(int a, int b) => a * b;  
double powN(double a, double n) => pow(a, n).toDouble();  
  
List<String> splitString(String line, String splitter){  
  return line.split(splitter);  
}  
  
String stringToLowerCase(String line){
```

```
    return line.toLowerCase();
}
```

```
String stringToUpperCase(String line){
    return line.toUpperCase();
}
```

```
String deleteSurroundingSpaces(String line) => line.trim();
```

Следующим действием в директории `test` текущего проекта создадим файл `my_functions_test.dart` и определим функцию верхнего уровня `main`, в которой и будут писаться тесты.

Для задания теста используется функция верхнего уровня `test`, в которую в качестве первого аргумента передается описание проводимого тестирования, а вторым аргументом выступает анонимная функция, вызываемая при выполнении самого теста. Тело анонимной функции должно заканчиваться тестовым утверждением `expect` в который передается результат работы функции, метода класса и т. д., а также ожидаемый результат выполнения вычисления:

```
import 'package:test/test.dart';

import 'package:my_project/my_functions.dart';

void main(){
    test('Проверка сложения', (){
        expect(add(3, 7), equals(10));
    });

    test('Удаление окружающих пробелов', (){
        var line = ' oO ';
        expect(deleteSurroundingSpaces(line), equals('oO'));
    });
}
```

Тесты могут быть сгруппированы. Для этого используется функция `group`, первым аргументом в которую передается описание группы, а вторым выступает анонимная функция, куда помещаются тесты. Внесем следующие изменения в файл `my_functions_test.dart`:

```
import 'package:test/test.dart';
```



```

import 'package:my_project/my_functions.dart';

void main() {
  group('Арифметические функции', () {
    test('Проверка сложения', () {
      expect(add(3, 7), equals(10));
    });
    test('Проверка умножения', () {
      expect(mul(3, 7), equals(21));
    });
  });

  group('Функции для работы со строками', () {
    test('Удаление окружающих пробелов', () {
      var line = ' оО ';
      expect(deleteSurroundingSpaces(line), equals('оО'));
    });
    test('Перевод в нижний регистр', () {
      var line = 'ПроВерка';
      expect(stringToLowerCase(line), equals('проверка'));
    });
  });
}

```

Когда необходимо выполнять какие-то действия перед тестом или освобождать ресурсы после его выполнения, на помощь приходят такие функции, как `setUp` и `tearDown`. Их необходимо использовать в начале функции `main` или группы тестов. В эти функции верхнего уровня передается анонимная функция, которая в случае с `setUp` будет выполняться перед каждым тестом в группе или набором тестов. А анонимная функция, передаваемая в `tearDown`, будет выполняться после теста, даже если он не будет пройден:

```

group('Арифметические функции', () {
  late int a;
  late int b;
  setUp(() {
    a = 3;
    b = 7;
  });
}

```

```
test('Проверка сложения', () {
  expect(add(a, b), equals(a+b));
});
test('Проверка умножения', () {
  expect(mul(a, b), equals(a*b));
});
});
```

Анонимные функции, в которых прописывается тест, могут быть отмечены ключевым словом `async`. Это позволяет тестировать асинхронную функциональность. Так, например, тест не будет считаться завершенным, пока `Future` не вернет значение:

```
test('Проверка отложенного умножения', () async {
  var value = await Future.value(mul(a, b));
  expect(value, equals(a*b));
});
```

10.3 Запуск тестов

Запускать тесты можно двумя способами: через команду терминала или используя встроенные средства VS Code. При запуске тестов через терминал имеется возможность запустить единственный файл на тестирование или все файлы с тестовым окружением проекта, расположенные в определенной директории. Для этого используется следующая команда: `dart run test путь_до_директории_с_тестами`. Если выполнять запуск тестирования из терминала проекта в VS Code, то команда для запуска тестов будет выглядеть следующим образом:

```
F:\code\dart\my_project>dart run test test
```

Во втором случае необходимо перейти в соответствующую панель и запустить тесты:

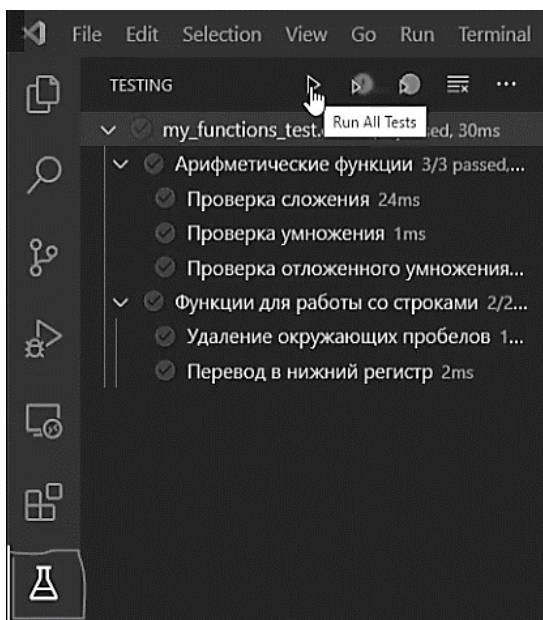


Рисунок 10.1 – Запуск тестов средствами VS Code

Давайте изменим проверяемую функцию сложения таким образом, чтобы она возвращала не корректное значение (к результату операции прибавляется еще значение 1) и повторно запустим тесты:

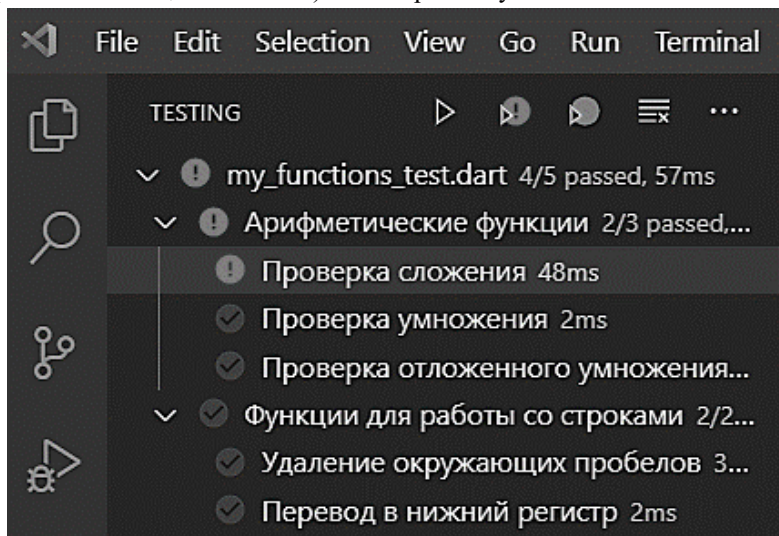


Рисунок 10.2 – Тест функции сложения не пройден

10.4 Конфигурация тестов

Порой проверяемая функциональность еще не реализована, а перед глазами маячит оповещение о том, что проверяющие ее тесты не пройдены. В этом случае, до момента ее реализации, файл с тестовым набором можно отметить аннотацией `@Skip('Описание почему пропускается')`, чтобы он пропускался при очередном запуске тестов проекта:

```
@Skip('Часть функционала не реализована')
// @Skip обязательно прописывается в самой первой строке файла
import 'package:test/test.dart';
import 'package:my_project/my_functions.dart';

void main() {
  // код тестов
}
```

Также имеется возможность пропускать отдельные тесты или их группы, передав им на вход именованный параметр `skip` (можно просто установить, как `true`, но лучше написать почему тест пропускается):

```
import 'package:test/test.dart';
import 'package:my_project/my_functions.dart';

void main() {
  group('Арифметические функции', () {
    late int a;
    late int b;
    setUp(){
      a = 3;
      b = 7;
    };
    test('Проверка сложения', () {
      expect(add(a, b), equals(a+b));
    }, skip: 'В левой пятке зачесалось!');
    test('Проверка умножения', () {
      expect(mul(a, b), equals(a*b));
    });
    test('Проверка отложенного умножения', () async {
      var value = await Future.value(mul(a, b));
    });
  });
}
```

```

    expect(value, equals(a*b));
  });
});

group('Функции для работы со строками', () {
  test('Удаление окружающих пробелов', () {
    var line = ' oO ';
    expect(deleteSurroundingSpaces(line), equals('oO'));
  });
  test('Перевод в нижний регистр', () {
    var line = 'ПроВерка';
    expect(stringToLowerCase(line), equals('проверка'));
  });
}, skip: 'Еще в разработке');
}

```

Теперь при запуске тестов проекта часть из них будет отмечена как пропущенные:

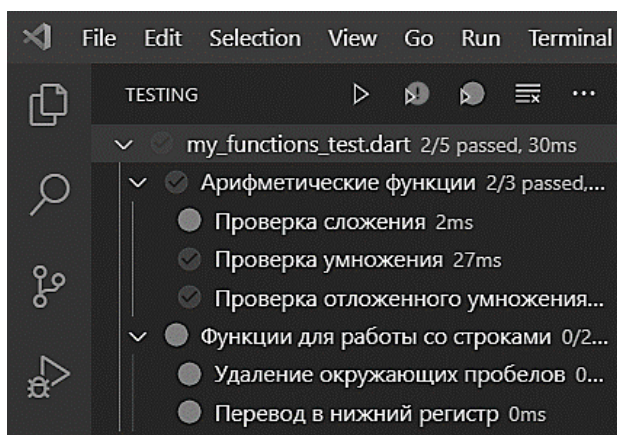


Рисунок 10.3 – Пропуск тестов

В ряде случаев может потребоваться настроить тест или тестовое окружение под различные платформы. Это может быть связано как с ограничениями самой платформы, так и с ее особенностями. Если необходимо задать настройку для всего файла, в его начале используйте аннотацию `@OnPlatform({'платформа': настраиваемый параметр})`. Либо же в группу или сам тест передавайте аргумент `onPlatform`:

```
import 'package:test/test.dart';
```

```

import 'package:my_project/my_functions.dart';

void main() {
  group('Арифметические функции', () {
    late int a;
    late int b;
    setUp(() {
      a = 3;
      b = 7;
    });
    test('Проверка сложения', () {
      expect(add(a, b), equals(a + b));
    }, onPlatform: {'windows': Skip('В левой пятке зачесалось!')});
    test('Проверка умножения', () {
      expect(mul(a, b), equals(a * b));
    });
    test('Проверка отложенного умножения', () async {
      var value = await Future.value(mul(a, b));
      expect(value, equals(a * b));
    });
  });

  group('Функции для работы со строками', () {
    test('Удаление окружающих пробелов', () {
      var line = ' оО ';
      expect(deleteSurroundingSpaces(line), equals('оО'));
    });
    test('Перевод в нижний регистр', () {
      var line = 'ПроВерка';
      expect(stringToLowerCase(line), equals('проверка'));
    });
  }, onPlatform: {'linux': Skip('В левой пятке зачесалось!')});
}

```

Хорошим подспорьем в настройке конфигураций тестов под различные платформы являются теги. Ими можно отметить, как сам файл (используя аннотацию `@Tags(['имя_тега'])`), так и отдельные тесты или группу тестов, задав значение именованному аргументу `tags`. Для примера отметим часть тестов меткой `'windows'`:

```

import 'package:test/test.dart';

```

```

import 'package:my_project/my_functions.dart';

void main() {
  group('Арифметические функции', () {
    late int a;
    late int b;
    setUp(() {
      a = 3;
      b = 7;
    });
    test('Проверка сложения', () {
      expect(add(a, b), equals(a + b));
    });
    test('Проверка умножения', () {
      expect(mul(a, b), equals(a * b));
    }, tags: ['windows']);
    test('Проверка отложенного умножения', () async {
      var value = await Future.value(mul(a, b));
      expect(value, equals(a * b));
    });
  });

  group('Функции для работы со строками', () {
    test('Удаление окружающих пробелов', () {
      var line = ' oO ';
      expect(deleteSurroundingSpaces(line), equals('oO'));
    });
    test('Перевод в нижний регистр', () {
      var line = 'ПроВерка';
      expect(stringToLowerCase(line), equals('проверка'));
    });
  }, tags: ['windows']);
}

```

Чтобы запустить тесты, отмеченные тегом, введите в терминале следующую команду: `dart run test test --tags "windows"`

```

F:\code\dart\my_project>dart run test test --tags "windows"
00:01 +3: All tests passed!

```

Рисунок 10.4 – Результат выполнения команды

Если имеется необходимость запускать только те тесты, которые не отмечены тегом, используйте для этого флаг `--exclude-tags` или `-x`: `dart run test test -x "windows"`

```
F:\code\dart\my_project>dart run test test -x "windows"
00:01 +0 -1: test\my_functions_test.dart: Арифметические функции Проверка сложения [E]
  Expected: <10>
  Actual: <11>

package:test_api          expect
test\my_functions_test.dart 14:7  main.<fn>.<fn>

00:01 +0 -1: loading test\my_functions_test.dart
Consider enabling the flag chain-stack-traces to receive more detailed exceptions.
For example, 'pub run test --chain-stack-traces'.
00:01 +1 -1: Some tests failed.
```

Рисунок 10.5 – Результат выполнения команды

Также имеется возможность задавать собственные теги. Только в этом случае их необходимо явно прописать в конфигурационном файле тестового окружения проекта `dart_test.yaml`, создав его в корневом каталоге проекта:

```
tags:
  super_test:
  slow_test:
```

Теперь заменим тег `'windows'` на `'super_test'` и запустим только те тесты, которые отмечены этим тегом: `dart run test test -t "super_test"`

```
F:\code\dart\my_project>dart run test test -t "super_test"
00:01 +3: All tests passed!
```

Рисунок 10.6 – Результат выполнения команды

Для более подробного знакомства с пакетом `test` и его возможностями обратитесь к документации [22]. С информацией по оформлению конфигурационного файла тестового окружения проекта можно ознакомиться в [23].

10.5 Использование пакета `mockito` для создания фиктивных объектов при проведении тестирования

Представим такую ситуацию, что у нас имеется удаленный сервер с REST API, который по запросу может возвращать нам данные о студенте в json-формате. И чтобы не организовывать каждый раз к нему кучу обращений при проведении тестирования (либо у нас нет возможности

это сделать с того компьютера, где проводится тестирование), проще всего, так как мы знаем, какие данные сервер будет возвращать, написать фиктивный объект и уже использовать при проведении тестов.

Для начала создадим новый консольный проект `students_server` и внесем изменения в файл `pubspec.yaml`, задав следующие зависимости проекта:

```
environment:  
  sdk: '>=2.12.0 <3.0.0'
```

```
dependencies:  
  http: ^0.13.3
```

```
dev_dependencies:  
  pedantic: ^1.10.0  
  test: ^1.17.3  
  mockito: ^5.0.7  
  build_runner: ^2.0.2
```

Следующим действием в файле `students_server.dart` каталога `lib` напомним следующий код:

```
import 'dart:convert';  
  
import 'package:http/http.dart' as http;  
  
class Student {  
  final String name;  
  late int age;  
  late int course;  
  
  Student(  
    {required this.name,  
    required this.age,  
    required this.course});  
  
  Student.fromJson(Map<String, dynamic> json)  
    : name = json['name'] {  
    age = json['age'];  
    course = json['course'];  
  }  
}
```

```

@override
String toString() {
  var student = 'Студент {имя: $name, возраст: $age, ';
  student += 'курс: $course}';
  return student;
}

Map<String, dynamic> toJson() {
  return <String, dynamic>{
    'name': name,
    'age': age,
    'course': course
  };
}
}

Future<Student> fetchStudent(http.Client client, int id) async {

  final response = await client.get(Uri.https('www.students-db.org',
    '/students', {'q': id.toString()}));

  if (response.statusCode == 200) {
    return Student.fromJson(json.decode(response.body));
  } else {
    throw Exception('Данные о студенте не загружены');
  }
}

```

Измените содержимое файла `students_server.dart` каталога `bin` следующим образом:

```
void main(List<String> arguments) { }
```

Далее в каталоге `test` создаем файл `fetch_student_test.dart` и приступаем к предварительному заполнению теста. То есть сделаем заготовку, на основе которой запустим кодогенератор для создания фиктивного объекта:

```

import 'package:test/test.dart';
import 'package:http/http.dart' as http;
import 'package:mockito/annotations.dart';

```

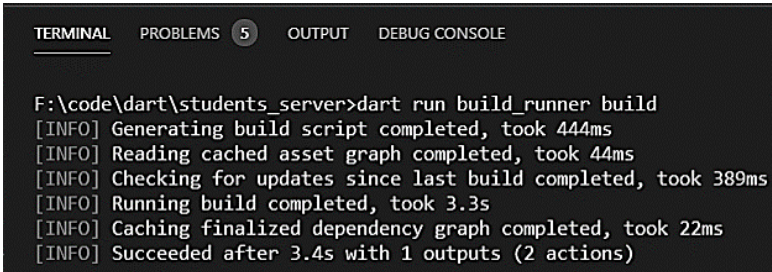
```
import 'package:mockito/mockito.dart';

import '../lib/students_server.dart';
// файл, который будет сгенерирован
import 'fetch_student_test.mocks.dart';

// аннотация для указания на основе какого
// класса генерировать фиктивный объект
@GenerateMocks([http.Client])
void main() {
}
```

Итак, настала пора вспомнить команду для запуска пакета `build_runner`, который автоматически сгенерирует код фиктивного объекта и поместит его в файл `fetch_student_test.mocks.dart`:

```
dart run build_runner build
```



```

F:\code\dart\students_server>dart run build_runner build
[INFO] Generating build script completed, took 444ms
[INFO] Reading cached asset graph completed, took 44ms
[INFO] Checking for updates since last build completed, took 389ms
[INFO] Running build completed, took 3.3s
[INFO] Caching finalized dependency graph completed, took 22ms
[INFO] Succeeded after 3.4s with 1 outputs (2 actions)

```

Рисунок 10.7 – Кодогенерация фиктивного объекта

После того, как файл с фиктивным объектом успешно сгенерировался, приступим к написанию тестов. Для этого в функцию верхнего уровня `main` файла `fetch_student_test.dart` добавим следующий код:

```
final id = 10;
late http.Client client;

setUp(() {
  // Create mock object.
  client = MockClient();
});
```

```
test('Получение данных о студенте и проверка их распаковки',
```

```

()async {
    // описываем поведение метода get у фиктивного объекта

    when(client.get(Uri.https(' www.students-db.org',
        '/students', {'q': id.toString()})))
        .thenAnswer((_) async =>
            http.Response('{ "name": "Alex", "age": 19, "course": 1}', 200));
    // Проверка возвращается ли экземпляр класса Student
    var student = await fetchStudent(client, id);
    expect(student, isA<Student>());
    // Корректно ли имя студента?
    expect(student.name, 'Alex');
    //Возраст?
    expect(student.age, 19);
    //Курс?
    expect(student.course, 1);
});

test('Проверка нештатной ситуации',
()async {
    when(client.get(Uri.https(' www.students-db.org',
        '/students', {'q': id.toString()})))
        .thenAnswer((_) async =>
            http.Response('Ошибка на стороне сервера', 500));
    expect(() => fetchStudent(client, id), throwsArgumentError);
});

```

Все готово для запуска теста с использованием фиктивного объекта `MockClient`:

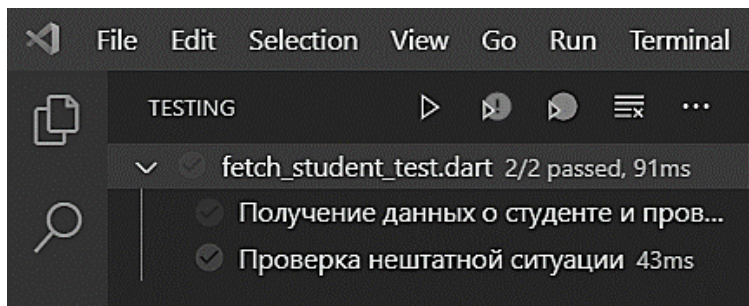


Рисунок 10.8 – Запуск теста с `MockClient`

Чтобы более подробно ознакомиться с возможностями и тонкостями использования пакета `mockito`, обратитесь к следующим ресурсам [24, 25].

Резюме по разделу

В данной теме мы рассмотрели какие инструменты необходимо использовать для тестирования функциональности разрабатываемого проекта. Конечно, были затронуты не все возможности таких пакетов Dart, как `test` и `mockito`, но и этого уже хватит для написания базовых тестов.

Писать тесты или не писать, решение за вами (или вашим руководителем). В любом случае от их наличия выиграют все, так как большинство ошибок будут отлавливаться на стадии разработки, что скажется на удобстве использования разрабатываемого приложения.

Если же у вас появилась мысль по поводу написания собственного пакета и загрузки его в `pub`, то одним из обязательных условий является наличие у него тестов.

Вопросы для самопроверки

1. Зачем писать тесты?
2. Что такое TDD? Какая ключевая идея лежит в её основе?
3. Какие виды тестирования используются в Dart?
4. Для чего используется пакет `test`? Приведите пример.
5. Каким образом можно конфигурировать тесты?
6. Для каких случаев следует использовать пакет `mockito`?

Упражнения

Напишите тесты к функциям и классам, реализованным в качестве упражнений в темах 3, 5 и 7.

Список используемых источников

1. Dart. URL: <https://dart.dev>
2. Sound null safety. URL: <https://dart.dev/null-safety>
3. Flutter. URL: <https://flutter.dev>
4. String class. URL: <https://api.dart.dev/stable/2.10.5/dart-core/String-class.html>
5. List<E> class. URL: <https://api.dart.dev/stable/2.10.5/dart-core/List-class.html>
6. Effective Dart: Documentation. URL: <https://dart.dev/guides/language/effective-dart/documentation>
7. Sanjib Sinha Quick Start Guide to Dart Programming: Create High-Performance Applications for the Web and Mobile // Apress; 1st ed. edition (November 30, 2019), p. 233
8. Eric Windmill Flutter in Action // Manning Publications; 1st edition (December 10, 2019), p. 368
9. Gilad Bracha, Erik Meijer The Dart Programming Language // Addison-Wesley Professional; 1st edition (December 20, 2015), p. 201
10. Creating packages. URL: <https://dart.dev/guides/libraries/create-library-packages3>
11. Package dependencies. URL: <https://dart.dev/tools/pub/dependencies>
12. Введение в JSON. URL: <http://www.json.org/json-ru.html>
13. json_serializable. URL: https://pub.dev/packages/json_serializable/install
14. The Event Loop and Dart. URL: <https://web.archive.org/web/20170704074724/https://webdev.dartlang.org/articles/performance/event-loop>
15. Futures - Isolates - Event Loop. URL: <https://www.didierboelens.com/2019/01/futures-isolates-event-loop/>
16. Future<T> class. URL: <https://api.dart.dev/stable/2.12.4/dart-async/Future-class.html>
17. Streams and Sinks in Dart and Flutter. URL: <https://dart.academy/streams-and-sinks-in-dart-and-flutter/>
18. Streams: Asynchronous Programming with Dart. URL: <https://ptyagicodecamp.github.io/streams-asynchronous-programming-with-dart.html>

19. Dart asynchronous programming: Isolates and event loops. URL: <https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bffc3e296a6a>
20. Write HTTP clients & servers. URL: <https://dart.dev/tutorials/server/httpserver>
21. Dart testing. URL: <https://dart.dev/guides/testing>
22. test. URL: <https://pub.dev/packages/test#running-tests>
23. Test package configuration file. URL: <https://github.com/dart-lang/test/blob/master/pkgs/test/doc/configuration.md#configuring-platforms>
24. mockito. URL: <https://pub.dev/packages/mockito>
25. mockito NULL_SAFETY_README. URL: https://github.com/dart-lang/mockito/blob/master/NULL_SAFETY_README.md

