Программирование на Dart
(для любопытных)

Dart компилируемый, объектно-ориентированный язык со статической типизацией.

1. Getting started

Традиционная helloworld программа:

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

Значит, программа должна содержать функцию ***main***, определяющую точку входа. Тело функции заключается в фигурные скобки, в конце строк обязательна точка с запятой. Стрелкой (→) я везде буду заменять слова получим, будет выведено и тому подобное. При копировании кода для выполнения стрелку с последующим текстом надо удалить.

Код программы надо поместить в файл с расширением имени ***.dart***. Назовём файл ***prog.dart***. Дальше я все программы примеров буду одинаково называть этим именем, не сохраняя их. Советую и читателю не сохранять эти ничтожно малые программы в памяти, а при желании их выполнить просто копировать код из этого текста.

Для выполнения программы из командной строки надо перейти в ту папку, где расположен файл и выполнить команду:

```
dart prog.dart
```

Для получения рабочего exe-файла надо выполнить команду:

```
dart2native prog.dart -o progapp.exe
```

Здесь ***progapp.exe*** название рабочего файла. Можно указать для него полный путь, если желательно хранить рабочие файлы не там, где исходные программы, а в другом месте. Рабочую программу можно запустить командой:

```
progapp
```

Ключ ***-o*** указывает, что результат компиляции надо записать в указанный после ключа файл. Вместо ***-o*** могут быть и другие ключи. Для компиляции dart-программ есть также команда ***compile*** с

дополнительными возможностями. Подробнее о компиляции программ на Dart далее поговорим отдельно.

По адресу https://dartpad.dartlang.org/ можно вызвать онлайн-редактор, который предоставляет дополнительные удобства для работы с dart-программами. Там есть также несколько типичных примеров для иллюстрации.

Посмотрим на ещё один пример для общего знакомства:

```
//Это комментарий
void f(int x) {
    print('The number is $x.');
}
void main() {
    var n = 42;
    f(n);  → The number is 42
}
```

На этом примере мы можем видеть, что:

- Строка с комментарием начинается со знака //. Многострочные комментарии помещаются между ограничителями /* и */. Комментарии документирования начинаются со знака ///. Дальше в примерах я почти никогда не буду применять комментарии, поскольку считаю, что при изучении языка код примеров должен быть максимально кратким, а необходимые пояснения удобнее делать в тексте.
- При объявлении функции указывается тип возвращаемого результата. Тип ***void*** означает отсутствие результата.
- При выводе текста использована интерполяция: текст ***$x*** означает, что вместо символа ***x*** будет выведено значение переменной ***x***. Вместо ***x*** может быть выражение и тогда его надо заключить в фигурные скобки, например, ***${x+3}*** и тогда получим: The number is 45. Интерполяция выполняется как при одиночных, так и при двойных кавычках.
- Обязательная функция ***main()*** может находиться в любом месте программы, а порядок объявления функций безразличен.
- Переменные объявляются со служебным словом ***var***, далее значения переменной можно изменять, например: ***n = 77;***

Для общего знакомства укажем здесь на некоторые важные особенности языка, которые дальше будут рассматриваться подробно:

- В Dart все сущности (terms) представляют экземпляры (объекты) каких-нибудь классов. При этом все объекты наследуют базовому классу ***Object***.
- Хотя Dart строго типизированный язык, объявление типа не всегда обязательно, так как имеется способность вывода (infer) типа из контекста.
- Для любого типа можно задать, может ли тип принимать особое значение ***null***. Эта способность в документации называется ***nullable*** (в языке Swift это называется optional). Чтобы сделать переменную nullable достаточно при указании её типа добавить знак вопроса, например ***int?***. В языке Swift эту операцию называют словом wrap. Обратное действие (unwrap) обозначается восклицательным знаком, который добавляется к идентификатору или к выражению.
- Функция высшего уровня (например main()) всегда привязана к классу (тогда она ***static***, обычно их называют методами класса) или к объекту (тогда она ***instance method*** – метод экземпляра). Допустимо создавать вложенные (локальные) функции внутри других функций.
- Соответственно, и переменные могут быть связаны с классом (static – переменная класса) или с объектом (instance variable – переменная экземпляра). Последние часто называют полями или свойствами.
- Dart поддерживает генерики (generic types), например List<int> - список целых, или List<Object> - список объектов любого типа.
- Dart не использует ключевые слова public, protected и private. Если идентификатор начинается со знака подчёркивания, то этот term будет private для своей библиотеки, подробности далее.
- Dart различает выражения (expressions), которые возвращают значение, и утверждения (statements), ничего не возвращающие. Statement может содержать один или несколько expression, но expression не может непосредственно содержать statement.
- В Dart все ключевые слова подразделяются на три типа; некоторые из них можно использовать для своих идентификаторов, другие — нельзя, а третьи можно использовать частично (не будем рассматривать подробно).

2. Переменные

Переменные объявляются со словом ***var*** и объявление можно совместить с инициализацией:

var x = "Hello!";

При этом тип переменной ***x*** будет выведен компилятором к значению ***String***. При объявлении переменной тип можно задавать и явно:

String x = "Hello!";

Теперь слово var не требуется. (Почему-то тип String записывается с большой буквы, тогда как int, double, bool – с маленькой). Между текстом в двойных и одинарных кавычках в данном случае нет разницы.

Согласно стилю Dart рекомендуется отдавать предпочтение объявлению переменных со словом ***var*** всюду, где это возможно, а не указывать тип явно.

Переменную можно объявить без инициализации:

var x;

или с объявлением типа.

double x;

Но такую переменную нельзя использовать, пока она не будет инициализирована. В первом случае её можно инициализировать любым значением, во втором — только числом с плавающей точкой. Только nullable переменные получают значение null при любом типе и их можно использовать:

double? x;

print(x == null); → true

Для подобной проверки в Dart есть удобный оператор ***assert***, но для его использования программу надо запускать с ключом ***enable-asserts***:

dart --enable-asserts prog.dart

Тогда можно вставить строку:

assert(x == null);

Если условное выражение даёт ***true***, эта строка просто игнорируется; если — ***false***, генерируется исключение. К условному выражению через запятую можно добавить текст, который будет выведен на экран:

assert(x == 99, "Условие даёт false");

При объявлении глобальной переменной есть некоторые нюансы:

```
int x;
void main() {
        x = 77;
        print(x);
}
```

Этот код не будет работать, требуется объявить переменную ***x*** как nullable:

```
int? x;
```

Или объявить её со словом ***var***:

```
var x;
```

Кроме того, имеется модификатор ***late***, который применяется в двух случаях:

- объявляется non-nullable глобальная переменная раньше её инициализации, так, как было в нашем примере.
- объявляется ленивая (lazily) переменная, которая не будет вычисляться до тех пор, пока она не будет использована.

```
late int x;
void main() {
        x = 77;
        print(x);
}
```

Этот код будет нормально работать.

Ленивые переменные иногда рационально использовать при громоздких (дорогих) вычислениях. Например, если запрограммировать вызов функции так:

```
late double x = f();
```

то переменная ***x*** не будет вычисляться до её использования, например, до строки ***print(x);***. Это полезно в том случае, когда функция ***f*** сложная и её вычисление занимает много времени и при этом есть вероятность, что переменная ***x*** может в дальнейшем не потребоваться вовсе. Проверим на примере:

```
import "dart:math";
void main() {
        print('hello');
        late double x = f();
//      print(x);
}
```

```
double f() {
        print('x вычислили');
        return sin(0.5);
}
```

В таком варианте наша программа выведет только слово hello. Но если раскомментировать строку

```
//      print(x);
```

то получим:

hello
x вычислили
0.479425538604203

Для не изменяемых (immutable) переменных применяется два модификатора: ***final*** или ***cons***t. Модификаторы заменяют слово ***var***:

```
final name = 'Катя';
const x = 1000000;
```

или они добавляются к указателю типа:

```
final double y = 2.75;
const double atm = 1.01325 * bar;
```

Между ***final*** и ***const*** есть некоторая разница. Для immutable переменных экземпляра применяется только модификатор ***final***. Если же константа объявляется на уровне класса (переменная класса), то она объявляется как ***const*** с дополнительным модификатором ***static***:

```
static const double x = 10.25;
```

Далее мы рассмотрим эти детали более подробно.

В большинстве случаев константа ***const*** одновременно неявно является и ***final***.

3. Базовые (встроенные) типы

К базовым в Dart относятся следующие типы:

- ***int***, ***double*** типы чисел, оба они являются подтипами типа ***num***. Переменные, объявленные с указателем типа ***num***, могут принимать и целые числа и числа с плавающей точкой.
- ***String*** тип строковых переменных, которые могут инициироваться строкой в двойных или одинарных кавычках.
- ***bool*** тип логических переменных с двумя возможными значениями: ***true*** и ***false***.

- ***List*** список (массив). Список ***m*** можно объявить, например, так:
List<num> m = [2, 3, 7.25];
- ***Set*** множество.
- ***Map*** ассоциированный список, словарь (dictionary), для краткости дальше будем называть их словом хеш.
- ***Runes*** часто называют также ***characters API***. Этот тип позволяет использовать разные нестандартные знаки.
- ***Symbol*** специальная форма для идентификаторов.
- ***Null*** имеет единственное значение ***null***.

Поскольку каждая переменная в Dart ссылается на объект — экземпляр класса, то можно говорить, что при инициализации переменной применяется конструктор класса. Некоторые базовые типы имеют свой явный конструктор, например, для создания хеша используется конструктор ***Map()***.

Среди базовых есть также типы, которые имеют специальное назначение:
- ***Object*** является суперклассом для всех других классов, кроме ***Null***.
- ***Future*** (или ***Stream***) используется для асинхронных действий.
- ***Iterable*** применяется в циклах.
- ***Never*** используется для функций, которые никогда не финишируют успешно (возвращают исключение).
- ***dynamic*** позволяет запрещать статическую (на этапе компиляции) проверку типов.
- ***void*** для никогда не используемых значений, чаще всего, как возвращаемый функцией тип.
- ***NaN*** это особое значение, которое возвращают операторы или функции, если результат отсутствует, например:
print(log(-6)); → NaN

Рассмотрим теперь базовые типы более подробно.

Числа

Тип ***int*** представляет целые числа в двоичном виде не длиннее, чем 64 бит (от 2^{-63} до 2^{63}). Тип ***double*** представляет 64-битные числа с плавающей точкой (в Dart отсутствует тип float). Оба типа ***int*** и ***double*** подтипы родительского типа ***num***, который имеет базовые операторы **+** (сложение), **-** (вычитание), ***** (умножение) и **/** (деление). К

этому же классу относятся такие базовые методы, как ***abs()*** - абсолютное значение, ***ceil()*** - ближайшее большее целое, ***floor()*** - ближайшее меньшее целое, ***round()*** - округление до целого. Их применение имеет такой синтаксис:

```
var x = 3.75;
var y = x.floor();
print(y); → 3
var z = x.round();
print(z); → 4
```

Математические функции находятся в модуле ***dart:math***, который надо предварительно импортировать:

```
import 'dart:math';
void main() {
    var x = 0.9;
    print(sin(x)); → 0.7833269096274834
}
```

Как обычно, можно писать и

```
import "dart:math";
```

Шестнадцатеричные числа принадлежат типу ***int***, при их записи впереди ставятся знаки ***0x***. В этом же классе находятся битовые операции, например ***<<()*** - сдвиг двоичного кода влево:

```
var x = 1;
var y = x <<(3);
print(y); → 8
```

Перевод чисел ***int*** в ***double*** происходит автоматически там, где это требуется:

```
int x = 3;
print(x * 2.5); → 7.5
print(11/x); → 3.6666666666666665
```

Тип ***num*** содержит также специальное значение ***Infinity***, которое возвращается, например, при делении на нуль:

```
var x = 1 / 0;
print(x); → Infinity
```

Для преобразования текста в число применяется метод ***parse*** с таким синтаксисом:

```
var x = int.parse('27');
var y = double.parse('7.089');
```

Целое число преобразуется в строку с помощью метода ***toString()***:
String x = 25.toString();

Для чисел с плавающей точкой применяется метод ***toStringAsFixed(n)***, где целое число ***n*** указывает, сколько знаков надо сохранить после десятичной точки:
String y = 3.14159.toStringAsFixed(4);
print(y); → 3.1416

При этом выполняется округление числа.

Для получения случайных чисел имеется встроенный класс Random, имеющий методы ***nextDouble()***, ***nextInt()*** и ***nextBool()***, которые возвращают числа с плавающей запятой, целые числа и логические true или false соответственно. Покажем их применение на примере:

```
import 'dart:math';
void main() {
    var x = Random();
    double a = x.nextDouble(); // Между 0.0 и 1.0: [0, 1)
    int b = x.nextInt(10); // Между 0 и 9
    bool c = x.nextBool(); // true или false
    print(a); → 0.1469877423167556
    print(b); → 6
    print(c); → true
}
```

Запуская эту программу повторно, мы будем получать всё новые случайные числа и логические значения.

Строки

Для строк можно использовать одинарные или двойные кавычки. Оба варианта практически равнозначны, только для для разделителя (‘) в английском тексте приходится применять обратный экранирующий слеш, например:
String s = 'it\'s';
который при двойных кавычках не нужен:
String s = "it's";

Интерполяция в строках работает одинаково в обоих вариантах:
var x = 2.5;
String y = 'сумма = ${x + 3}';
String z = "сумма = ${x + 5}";
print(y); → сумма = 5.5
print(z); → сумма = 7.5

В Dart применяются обычные управляющие символы, например ***\n*** – перевод строки.

Для конкатенации строк применяется оператор суммирования (+):
var x = 'Мама';
print(x + ' мыла раму'); → Мама мыла раму

Текст, расположенный на нескольких строчках автоматически объединется в одну строку:
var x = "У лукоморья "
"дуб "
'зелёный';
print(x); → У лукоморья дуб зелёный

Переменные типа String можно инициировать текстом, расположенным на нескольких строках. При этом надо использовать тройные кавычки, одиночные или двойные безразлично:
String x = '''
Отговорила
роща
золотая''';
print(x); →

Отговорила
роща
золотая

Тройные кавычки не обязательно располагать на отдельных строках, как это принято в Swift.

Если перед текстом поставить букву ***r***, то в тексте не будут действовать управляющие символы:
var x = r'Бежит река \n в тумане тает';
print(x); → Бежит река \n в тумане тает

Такие строки в документации называются словом **raw**.

Bool

Для типа ***bool*** ограничимся приведением нескольких характерных примеров:

var x = '';
print(x.isEmpty); → true

Метод ***isEmpty*** позволяет узнать содержит ли переменная типа String какие-нибудь знаки, или она пустая. Этот же метод позволяет анализировать и коллекции, например списки:

var a = [1, 2];
print(a.isEmpty); → false

В Dart применяются обычные операторы сравнения: ==, >=, <=, !=, например:

var x = 0;
print(x <= 0); → true

Переменные nullable можно сравнивать со значением null

int? x;
print(x == null); → true

Метод ***isNaN*** проверяет переменные на значение ***NaN***, которое возвращают функции при исключительных ситуациях:

var x = 0 / 0;
print(x.isNaN); → true

Списки (List)

Список в Dart то же самое, что обычно называется массивом. Можно объявить пустой список без указания типа элементов:

var m = [];

В этом случае тип элементов списка ***m*** будет ***Object***, его можно указать и явно:

List<Object> m = [];

Такой список может содержать элементы любого типа:

m = [1, true, 'Anna', 25.089];

При этом, как видим, тип элементов списка указывается в угловых скобках. Это так называемый параметр типа, их мы позже рассмотрим подробно.

Dart не позволяет инициировать элементы с индексами больше, чем в объявленном списке, например, мы не можем написать:

m[4] = 77;

Компилятор выдаст: Invalid value: Not in inclusive range 0..3: 4

К списку можно добавлять другой список с помощью оперетора сложения (+), как и при конкатенации строк:

m = m + [77, 'Люда'];

print(m); → [1, true, Anna, 25.089, 77, Люда]

При этом допустима и краткая форма:

m += [77, 'Люда'];

Для добавления элемента к списку применяется функция ***add***:

m.add('Катя');

Если объявить список с одновременной инициализацией элементов, у которых один и тот же тип, то этот тип будет выведен компилятором:

var m = [1, 2, 3];

Это будет то же самое, как если бы мы объявили:

List<int> m = [1, 2, 3];

При этом список ***m*** может содержать элементы только типа ***int***:

m[1] = 9;

print(m); → [1, 9, 3]

Dart позволяет ставить запятую после последнего элемента в списке:

var m = [1, 2, 3,];

Иногда это позволяет избежать ошибки при вводе данных. Как говориться, мелочь, а приятно.

Для определения длины списка (числа элементов в списке) применяется метод ***length***:

print(m.length); → 3

Можно создавать не изменяемые (константные) списки:

var m = const [1, 2, 3];

Теперь нельзя изменять элементы списка:

m[1] = 9; - будет зафиксирована ошибка

Располагать элементы в списке можно столбцом:

```
var m = [
    'aa',
    'bb',
    'cc'
];
```

print(m); → [aa, bb, cc]

Dart позволяет использовать так называемый ***spread*** – оператор (spread – распространение, разрастание). Оператор spread обозначается тремя точками и позволяет добавлять список к другому списку непосредственно в квадратных скобках:

```
var m = [1, 2, 3];
var m1 = ['aa', 'bb', 'cc', ...m];
print(m1); → [aa, bb, cc, 1, 2, 3]
```

Есть даже разновидность этого оператора ***…?*** (null-aware – оператор), который учитывает возможность nullable членов в добавляемом списке (aware – знающий, осведомлённый).

Существует разновидность списков, называемых ***collection if***. Покажем на примере:

```
var x = 77;
var m = [1, 2, 3, if(x == 77) 4];
print(m); → [1, 2, 3, 4]
```

Здесь если ***x*** не будет равно ***77***, то элемент ***4*** не будет содержаться в списке.

Ещё одна разновидность списков называется ***collection for***:

```
var m = [2, 3, 4];
List<String> m1 = ['aa', 'bb', for(var i in m) '${i * i}'];
print(m1); → [aa, bb, 4, 9, 16]
```

Как видим, над элементами добавляемого списка можно попутно выполнять какие-то операции. Для приведения добавляемых элементов к типу String здесь использована интерполяция.

Иногда могут случаться неожиданные ошибки, например:

```
void f(List<int> m) => print(m);
void main() {
    var a = [];
    a.add(1);
    a.add(2);
    f(a);
}
```

Этот код не будет работать, так как для списка ***a*** будет выведен тип ***List<dynamic>***, тогда как аргументом функции ***f*** должен быть список типа ***List<int>***. В примере тело функции f добавлено через стрелку => и без фигурных скобок, такую форму обсудим позже.

Множества (Sets)

Множества (***Sets)*** это коллекции, содержащие не упорядоченные элементы без повторяющихся значений. Элементы множеств заключаются в фигурные скобки. В остальном множества мало отличаются от списков. Для объявления пустого множества можно применять два способа:

var s = <String>{};
print(s); → {}

Или:

Set<String> s = {};
print(s); → {}

Если объявить:

var s = {};

то будет создано не множество, а хеш с типом ***<dynamic, dynamic>***.

При объявлении с инициализацией, тип элементов выводится автоматически, так же, как и для списков:

var s = {'aa', 'bb', 'cc', 'bb', 'dd'};
print(s); → {aa, bb, cc, dd}

Повторяющиеся элементы игнорируются. Если множество содержит элементы разных типов, то тип самого множества будет Object.

Для добавления элементов в множество применяется метод ***add()***, а с помощью метода ***addAll()*** можно к множеству добавить другое множество:

v***ar s = <String>{};***
s.add('Люда');
Set<String> s1 = {'Катя', 'Лариса'};
s.addAll(s1);
print(s); → {Люда, Катя, Лариса}

Для определения числа элементов в множестве применяется всё тот же метод length:

print(s.length); → 3

Множество можно сделать константным:

Set<int> s = const {1, 5, 3};

Теперь к множеству ***s*** нельзя добавить новые члены.

Хеши (Maps)

Элементы хеша (ассоциированного списка, словаря — dictionary) представлены парами ключ — значение (key – value). Для ключей и значений можно использовать любые типы, базовые и пользовательские. Значения отделяются от ключей с помощью двоеточия. Пустой хеш можно объявить так:

var h = {};

Теперь его можно инициировать:

h = {1: 'aa', 2: 'bb', 3: 'cc'};
print(h); → {1: aa, 2: bb, 3: cc}

Можно с указанием типа для ключей и значений:

Map<int, String> h = {1: 'aa', 2: 'bb', 3: 'cc'};

Или с применением конструктора Map:

var h = Map<int, String>();
h = {1: 'aa', 2: 'bb', 3: 'cc'};
print(h); → {1: aa, 2: bb, 3: cc}

Здесь слово ***Map*** обозначает конструктор для типа ***Map***, при этом использованы круглые скобки. Обычно при использовании конструктора используется слово ***new***. Здесь его можно тоже применить, но это не обязательно:

var h = new Map<int, String>();
h = {1: 'aa', 2: 'bb', 3: 'cc'};

При разнообразии типов для ключей и значений можно тип не указывать и тогда он будет выведен к типу ***<dynamic, dynamic>***:

var h = {1: 'aa', 'key': 0.75, 0.3: true};

Или можно указать такой тип явно:

Map<dynamic, dynamic> h = {1: 'Люда', 'key': 0.75, 0.3: true};

Всё будет нормально работать и при таком варианте:

Map<Object, Object> h = {1: 'Люда', 'key': 0.75, 0.3: true};

Элементы хеша создаются так же, как и для списков, только вместо индексов используются ключи:

h['Сергей'] = 58;
print(h); → {1: Люда, key: 0.75, 0.3: true, Сергей: 58}

По ключу можно извлечь значение:

print(h[0.3]); → true

При попытке извлечь значение для не существующего ключа получим ***null***:
print(h[77]); → null

Метод ***length*** возвращает размер хеша:
print(h.length); → 4

Хеши позволяют использовать spread - оператор (**…** и **...?**), а также коллекции ***if*** и ***for*** аналогично спискам.

Перечислители (Emums)

Перечислитель создаётся с помощью слова ***enum***:

```
enum Color { red, green, blue }
void main() {
    print(Color.green.index); → 1
    List<Color> m = Color.values;
    print(m[2]); → Color.blue
    var x = Color.blue;
    switch (x) {
        case Color.red: print('Red as roses!');
        break;
        case Color.green: print('Green as grass!');
        break;
        default: print(x); →Color.blue
    }
}
```

С помощью метода ***index*** можно получить номер (индекс) элемента. В примере перечислитель использован в переключателе ***switch — case*** (смотрите далее).

4. Функции

Функции в Dart также являются объектами и имеют тип ***Function***. Они могут служить значениями для переменных и быть аргументами для других функций:

```
import "dart:math";
void main() {
    Function g = f;
```

print(g(0.5)); → 0.958851077208406
}
double f(x) {return 2 * sin(x);}

Здесь переменная ***g***, имеющая тип ***Function***, инициирована функцией ***f***.

```
import "dart:math";
void main() {
    print(g(f)(0.5)); → 0.958851077208406
}
Function g(Function q) {return q;}
double f(x) {return 2 * sin(x);}
```

А в этом примере функция ***g*** имеет аргумент в виде функции ***q*** и возвращает эту функцию, как результат.

Указатель типа Function часто можно опускать так же, как мы это делаем для других типов. Например, выше мы могли бы написать:

```
Function g(q) {return q;}
```

Здесь тип аргумента ***q*** выводится из контекста.

Функции в качестве аргументов часто используются в итераторах, например:

```
void main() {
    var m = [1, 2, 3];
    m.forEach(f); → 1 2 3
}
void f(int x) { print(x); }
```

Здесь итератор ***forEach()*** принимает функцию ***f***, как аргумент.

Функции могут быть элементами списка или других коллекций, например:

```
void main() {
    var m = [f, g, q];
    m.forEach((t) { t(3); }); → 3 9 27
}
void f(int x) { print(x); }
void g(int x) { print(x * x); }
void q(int x) { print(x * x * x); }
```

Итератор ***forEach*** каждой функции из списка присваивает имя ***t*** и вызывает их с аргументом ***3***.

Для функций, тело которых состоит только из одного выражения, допустимо применять краткую форму со стрелкой => (обычно называют arrow – синтаксис). Например, функцию ***f*** в примере выше мы могли бы определить так:

double f(x) => 2 * sin(x);

При этом выражение справа от стрелки не надо заключать в фигурные скобки.

Аргументы функции можно сделать именованными, для этого список аргументов надо представить в виде множества (Set):

```
void main() {
    print(f(y: 3, x: 2.0));  → 5.0
}
double f({x, y}) => x + y;
```

Значит, при вызове функции имя аргумента отделяется от значения двоеточием. Польза от именованных аргументов заключается в том, что при вызове функции их можно располагать в произвольном порядке, что особенно удобно, если аргументов много.

Именованные аргументы автоматически являются и не обязательными (optional). Если при вызове функции значение аргументу не передаётся, он получает значение ***null***:

```
void main() {
    f(y: 3);  → null, 3
}
void f({x, y}) => print('$x, $y');
```

Если у не обязательных элементов указывается тип, то надо добавлять знак вопроса (аргумент должен быть nullable):

```
void main() {
    f(y: 3);  → null, 3
}
void f({int? x, int? y}) => print('$x, $y');
```

При этом все аргументы в списке должны быть nulable. А если надо указать, что для данного аргумента требуется задавать значение всегда, используется модификатор ***required***:

```
void main() {
    f(y: 3);  → null, 3
}
void f({int? x, required int y}) => print('$x, $y');
```

Если аргументы функции не являются именованными, то для того, чтобы сделать некоторые из них не обязательными достаточно заключить в квадратные скобки:

```
void main() {
      print(f('Anna', 'Marta'));  → Anna, Marta
}
String f(String x, String y, [String? z]) {
  var r = '$x, $y';
  if (z != null) {
    r = '$r and $z';
  }
  return r;
}
```

Здесь необязательный аргумент ***z*** получает значение ***null***. Если аргументу ***z*** передать значение, получим:

print(f('Anna', 'Marta', 'Peter')); → Anna, Marta and Peter

При объявлении функции значения её именованных аргументов можно задать «по умолчанию» с использованием знака равенства. При этом список аргументов надо также задать в формате множества (Set):

```
void main() {
      print(f(y: 9));  → 45
}
int f({int x = 5, int y = 7}) {
      return x * y;
}
```

Здесь заданное по умолчанию значение аргумента ***y***, равное ***7***, изменяется на ***9*** при вызове функции.

При таком варианте все аргументы функции должны быть заданы по умолчанию, нельзя, например, написать:

int f({int x = 5, int y}) {…};

Для не именованных аргументов значение по умолчанию можно задать с использованием квадратных скобок:

```
void main() {
      print(f(1, 2));  → 6
}
int f(x, y, [z = 3]) {
```

```
    return x * y * z;
}
```

Здесь можем применить, например, такой вызов функции ***f***:

print(f(1, 2, 7)); → 14

Аргументам по умолчанию можно присваивать с помощью знака равенства списки и хеши, которые при этом должны быть константными:

```
void main() {
    f();
}
void f ( {List<int> m = const [1, 2, 3],
Map<int, String> h = const {1: 'осёл', 2: 'петух',3: 'собака'}}) {
    print('m: $m');  → m: [1, 2, 3]
    print('h: $h');  → h: {1: осёл, 2: петух, 3: собака}
}
```

Любая функция в Dart возвращает значение. Если явно возвращаемый результат не указан с помощью оператора ***return***, то возвращается значение ***null***.

Функция main()

Функция ***main()*** может принимать аргумент в виде списка с элементами типа ***String***. Если при запуске программы в конце команды напечатать какие-то текстовые данные, разделяя их пробелом, то эти данные будут помещены в список — аргумент функции ***main()***. Этот список будет доступен в теле функции ***main()***. Например, если мы запустим программу такой командой

dart prog.dart 'Люда' 'Катя' 'Лариса'

```
void main(List<String>m) {
  print(m);  → ['Люда', 'Катя', 'Лариса']
}
```

то будет выведен полученный список. Кстати, кавычки у текстовых величин здесь можно опускать, то-есть, допустим и такой вызов: ***dart prog.dart Люда Катя Лариса***. Если напечатать здесь числа, то они автоматически будут представлены в виде текстовых величин. Поскольку тип элементов в массиве ***m*** всегда ***String*** по умолчанию, то тип можно и не указывать:

```
void main(m) {
  print(m);
}
```

Анонимные функции

Анонимные функции часто называют словом closure, впрочем, часто так называют и именованные функции, используемые в качестве аргументов. Фактически closure ничем не отличаются от обычных функций, за исключением того, что им не присваивается имя, когда в нём нет необходимости:

```
void main() {
      List<String> s = [];
      var m = ['apples', 'bananas', 'oranges'];
      m.forEach((t) { s.add(t.toUpperCase());});
      print(s); → [APPLES, BANANAS, ORANGES]
}
```

Здесь итератору forEach передаётся анонимная функция (closure) ***{ s.add(t.toUpperCase()); }***
которая к списку ***s*** добавляет элементы ***t***, переводя текст в верхний регистр. В общем виде анонимная функция в Dart выглядит примерно так: ***(аргументы) {блок}***, например:

```
(int x, y) { return x + y;}
```

Анонимной функцией можно инициировать переменную, которая становится обычной функцией:

```
void main() {
      print(f(2,3)); → 5
}
var f = (int x, y) { return x + y;};
```

Можно, конечно, применять и arrow – синтаксис:

```
void main() {
      print(f(2,3)); → 5
}
var f = (int x, y) => x + y;
```

Анонимная функция может быть результатом, возвращаемым другой функцией посредством оператора ***return***:

```
Function f(int x) {
```

```
        return (int i) => x + i;
}
void main() {
        var g1 = f(2); print(g1(3)); → 5
        var g2 = f(4); print(g2(3)); → 7
}
```

Функция ***f*** возвращает анонимную функцию, этим результатом инициированы переменные ***g1*** и ***g2***, которые сами становятся функциями.

Область видимости переменных

Внешние по отношению к функции переменные видны в теле функции без ограничения. На примере показана доступность переменных при наличии вложенных функций, а также способ вызова таких функций:

```
int x = 1;
void main() {
        var y = 2;
        void f1() {
                var z = 3;
                void f2() {
                        var u = 4;
                        print(x);
                        print(y);
                        print(z);
                        print(u);
                }
                f2();
        }
        f1(); → 1 2 3 4
}
```

На следующем примере показана доступность внешнего closure в теле функции:

```
Function f(int x) {
```

```
        return (int i) => x + i;
}
void main() {
        var f1 = f(2);
        var f2 = f(4);
        print(f1(3)); → 5
        print(f2(3)); → 7
}
```

Здесь внешняя функция ***f*** возвращает closure в качестве результата. В теле функции ***main*** имеется возможность использования этого closure без ограничения.

Сравнение функций на равенство

Поскольку функции являются объектами и похожи на переменные, то их, как переменные, можно сравнивать, например:

```
int f(int d) { return d * d;}
class A {
        static int s(int d) {return d * d;}
        int t(int d) {return d * d;}
}
void main() {
        Function x;
        x = f;
        print(f == x); → true
        x = A.s;
        print(A.s == x); → true
        var v = A();
        var w = A();
        var y = w;
        x = w.t;
        print(y.t == x); → true
        print(v.t == w.t); → false
}
```

Здесь объявлена внешняя функция ***f***, а в классе ***A*** объявлен статический метод ***s*** и метод экземпляра ***t***. Функция и оба метода имеют одинаковую функциональность. Созданы два экземпляра

класса ***A*** – ***v*** и ***w***. Выполнено сравнение с помощью оператора == нескольких функций и методов. Как видим, только методы двух разных экземпляров ***v.t*** и ***w.t*** оказались не равными друг другу. Когда будем рассматривать классы, обсудим детали подробнее.

Операторы

Dart имеет обычный набор операторов. Рассмотрим здесь только наиболее специфичные.

Арифметический оператор ~/ возвращает целую часть частного при делении целых чисел:

print(15 ~/ 4); → 3

Числа могут иметь тип ***double***, но результат всегда ***int***:

print(15.7 ~/ 2.9); → 5

Оператор ***%*** возвращает остаток при делении чисел:

print(17 % 3); → 2

Если числа double, результат тоже будет double:

print(16.7 % 2.3); → 0.6

Для проверки типов используются операторы ***as***, ***is*** и ***is!***. Оператор ***as*** при несоответствии указанного типа генерирует исключение. При этом применяется такой синтаксис:

```
class A { int x = 0; }
class B { int x = 0; }
void main() {
    var p = A();
    (p as A).x = 7;
    print(p.x); → 7
    var q = B();
    (q as A).x = 9; → type 'B' is not a subtype of type 'A' in type cast
    print(q.x);
}
```

Оператор ***is*** применяется с условным оператором ***if***, что позволяет избежать исключения. Если тип не соответствует указанному, блок просто пропускается:

```
class A { int x = 0; }
class B { int x = 0; }
void main() {
```

```
        var p = A();
        if (p is A) { p.x = 7; }
        print(p.x); → 7
        var q = B();
        if (q is A) { q.x = 9; }
        print(q.x); → 0
}
```

Оператор ***is!*** противоположен оператору ***is***:

```
if (q is! A) { q.x = 9; }
print(q.x); → 9
```

Кроме обычного оператора присваивания **=** есть ещё оператор ***??=***, который присваивает переменной новое значение только в том случае, когда до этого переменная имела значение ***null***. Если переменная имела другое значение (не null), то присваивание не выполняется и сохраняется старое значение.

```
var x = null;
x ??= 77;
print(x); → 77
x ??= 99;
print(x); → 77
```

5. Ветвления и циклы

Условный оператор ***if-else*** имеет обычный вид:

```
var x = 2;
if (x == 1) {var y = 77; print(y);} else {var y = 99; print(y);} → 99.
```

Ветвь ***else*** может отсутствовать.

Имеется краткая форма ***? - :***, выполняющая то же самое:

```
var x = 5;
x is String ? print('String') : print('no String'); → no String
```

Условный оператор возвращает значение:

```
var x = 'Hello';
var y = x is String ? 'String' : 'no String';
print(y); → String
```

Поэтому можно создавать, например, такие функции:

```
void main() {
        print(f(0.25)); → no String
```

```
}
String f(x) => x is String ? "String" : "no String";
```

Для nullable переменных есть краткая форма условного оператора:

```
void main() {
    print(f(null)); → 77
}
int f(int? x) => x ?? 77;
```

Здесь если ***x*** равно ***null***, возвращается значение справа от ***??***, если ***x*** равно целому числу, возвращается это число.

Так называемый «каскадный оператор», обозначаемый двумя точками, позволяет писать более «плавный код», как он назван в документации:

```
void main() {
    var p = A()
    ?..z = 5.7
    ..x = 42
    ..y = 'Hello';
    print(p.y); → Hello
}
class A {
    var x = 0;
    var y = '';
    double? z = null;
}
```

Для nullable поля ***z*** надо ставит знак вопроса перед точками. При этом этот оператор должен стоять впереди. Обращаю внимание на то, что точка с запятой должна стоять только в конце каскада. Более детально каскадный оператор обсудим при рассмотрении классов.

В Dart применяется обычный оператор цикла ***for***:

```
void main() {
    var m = [];
    for (var i = 2; i < 6; i++) { m.add(f(i)); }
    print(m); → [4, 9, 16, 25]
}
int f(x) { return x * x; }
```

При работе с коллекциями можно применять также разновидность оператора ***for*** со служебным словом ***in***:

```
void main() {
        var m = ["Luda", "Katja", "Larisa"];
        for (var x in m) {
                print(x); → Luda Katja Larisa
        }
}
```

Ранее мы уже использовали для циклов итератор ***forEach***, которому передаётся функция:

```
void main() {
        var m = [1, 2, 3];
        m.forEach((t){print(t);}); → 1 2 3
}
```

Или с использованием arrow – синтаксиса:

```
void main() {
        var m = [1, 2, 3];
        m.forEach((t) => print(t));
}
```

Иногда явно аргументы можно не передавать:

```
m.forEach(print);
```

Если надо получить новый список, придётся сделать так:

```
void main() {
        var s = [];
        var m = [1, 2, 3];
        m.forEach((t) => s.add(f(t)));
        print(s); → [1, 4, 9]
}
int f(x) {return x*x;}
```

Dart позволяет также применение цикла ***while***:

```
void main() {
        var x = 0;
        while(x <= 5) {print(x); x += 1;} → 0 1 2 3 4 5
}
```

А также и ***do – while***:

```
void main() {
        var x = 0;
        do { print(x); x += 1;}
        while(x <= 5);
```

```
}
```

Выйти из цикла можно с помощью оператора ***break***:

```
void main() {
	var x = 0;
	while (true) {
		if (x > 5) break;
		print(f(x)); x += 1;
	}
}
int f(x) {return x*x;}
```

Оператор ***continue*** позволяет вернуться на начало цикла из нужной точки:

```
void main() {
	var m = [1,2,3,4,5,6];
	for (int i = 0; i < m.length; i++) {
		var x = m[i];
		if (x < 4) { continue; }
		print(f(x));  →16 25 36
	}
}
int f(x) {return x*x;}
```

Dart позволяет использовать также такой «синтаксический сахар»:

```
void main() {
	var m = [1,2,3,4,5,6];
	m
	.where((t) => t >= 4)
	.forEach((t) => print(f(t)));
}
int f(x) {return x*x;}
```

Получим тот же результат, что и в предыдущем случае.
Здесь локальной переменной ***t*** последовательно передаются значения элементов списка ***m***. Смысл строки ***.where((t) => t >= 4)*** понятен из перевода слова ***where*** – где, когда, который. Локальную переменную во второй строчке можно обозначить не ***t***, а другой буквой, это разные переменные, например: ***.forEach((s) => print(f(s)));***

Переключатель ***switch — case*** имеет обычный вид, за исключением того, что каждое сопоставление с образцом ***case*** должно заканчиваться директивой ***break, continue***, ***throw***, или ***return***:

```
void main() {
	var x = 7;
	switch (x) {
		case 5 : print("x < 7");
		break;
		case 7 : print("x == 7");
		break;
		case 9 : print("x > 7");
		break;
		default : print("unknown");
	}
}
```

Пример программы для факториала числа с использованием переключателя и рекурсии:

```
int f(int x,int n) {
	switch (n) {
		case 1 : return x;
		default : {x = x * n; return f(x, (n-1));}
	}
}
void main() {
	print(f(1, 5)); → 120
}
```

Нюанс тут в том, что перед рекурсивным вызовом функции требуется директива ***return*** для того, чтобы обе ветви условного выражения возвращали результат одного и того же типа:

```
return f(x, (n-1));
```

Приведу ещё эту же программу с использованием условного оператора if-else:

```
int f(int x, int n) {
  if (n < 2) {return x;} else {x = x * n; return f(x, (n-1));}
}
void main() {
  print(f(1, 5)); → 120
```

}

6. Классы

Dart объектно-ориентированный язык, любой объект является экземпляром какого-либо класса. Все классы (кроме Null) наследуют классу Object.

Для объявления класса используется слово ***class*** после которого идёт название класса с большой буквы. В теле класса могут быть объявлены переменные и функции, которые по общепринятой терминологии называются переменными и методами экземпляра класса. Приведём пример класса ***Point***:

```
import 'dart:math';
class Point {
        double x = 0;
        double y = 0;
        Point(double x, double y) {
                this.x = x;
                this.y = y;
        }
        double f(a, b) {return sqrt(a*a + b*b);}
}
void main() {
        var p = Point(3, 4);
        print(p.x); → 3.0
        print(p.f(3, 4)); → 5.0
}
```

Класс ***Point*** имеет две переменных экземпляра класса ***x*** и ***y*** и метод экземпляра класса ***f***. Текст:

```
Point(double x, double y) {
                this.x = x;
                this.y = y;
        }
```

называется конструктором класса. Конструктор позволяет передать значения для переменных экземпляра при создании экземпляра класса:

var p = Point(3, 4);

Экземпляр класса создаёт функция ***new***, которую можно вызывать явно:

var p = new Point(3, 4);

Но Dart позволяет опускать ***new*** и тогда эта функция вызывается по умолчанию. Имеется синтаксический сахар, позволяющий применять более краткий код для конструктора:

```
import 'dart:math';
class Point {
        double x = 0;
        double y = 0;
        Point(this.x, this.y);
        double f(a, b) {return sqrt(a*a + b*b);}
}
```

В конструкторе можно добавить блок в фигурных скобках, который будет выполнен при создании экземпляра класса:

```
class Point {
        double x, y;
        Point(this.x, this.y)  { print('${x + y}');}
}
void main() {
        var p = Point(3, 4); → 7.0
}
```

Можно опускать объявление конструктора и тогда он применяется неявно, но при этом невозможна передача параметров:

```
import 'dart:math';
class Point {
        double x = 0;
        double y = 0;
        double f(a, b) {return sqrt(a*a + b*b);}
}
void main() {
        var p = Point();
        p.x = 7;
        print(p.x); → 7
        print(p.f(3, 4)); → 5
}
```

Конструктор класса может быть именованным, что позволяет применять несколько конструкторов в классе. Имя конструктора добавляется через точку:

```
class Point {
        double x = 0;
        double y = 0;
        Point(this.x, this.y);
        Point.origin();
}
void main() {
        var p = Point(3, 4);
        var p1 = Point.origin();
        print(p1.x); → 0
}
```

В этом примере использованы два конструктора: не именованный конструктор ***Point***, имеющий параметры и именованный конструктор ***Point.origin*** без параметров. Метод ***runtimeType*** позволяет узнать тип объекта на этапе runtime:

print(p.runtimeType); → Point

Метод ***runtimeType*** можно применять к любой сущности (term), поскольку в Dart всё имеет какой-либо тип:

```
import 'dart:math';
print(sin.runtimeType); → (num) => double
```

В данном случае метод runtimeType возвратил сигнатуру математической функции ***sin*** и значит тип функций представлен их сигнатурой. Функция ***sin*** принимает аргумент типа ***num*** и возвращает результат типа ***double***.

Переменным экземпляра можно задать значения непосредственно в конструкторе, перечисляя их через запятую после двоеточия (в документации это называется Initializer list):

```
class Point {
        double x = 0;
        double y = 0;
        Point() : x = 5, y = 7;
}
void main() {
        var p = Point();
```

```
        print('x = ${p.x}, y = ${p.y}');  → x = 5.0, y = 7.0
}
```

Методы экземпляра могут иметь аргументы, представленные экземпляром самого класса, например:

```
import 'dart:math';
class Point {
  double x = 0;
  double y = 0;
  Point(this.x, this.y);
  double f(Point r) {
    var dx = x - r.x;
    var dy = y - r.y;
    return sqrt(dx * dx + dy * dy);
  }
}
void main() {
        var p = Point(1, 2);
        var q = Point(4, 6);
        var d = p.f(q);
        print(d);  → 5.0
}
```

Здесь методу ***f*** назначен аргумент ***Point r***, представляющий экземпляр класса ***Point,*** а в теле функции использованы переменные экземпляра ***r.x*** и ***r.y***.

С помощью директивы ***operator*** можно изменять функциональность базовых операторов, в частности арифметических операторов. Например, можно операторы + и - использовать для сложения и вычитания алгебраических векторов.

```
class Vector {
        int x, y;
        Vector(this.x, this.y);
        Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
        Vector operator -(Vector v) => Vector(x - v.x, y - v.y);
}
void main() {
        var v = Vector(5, 7);
        var w = Vector(1, 2);
```

```
    var p1 = (v + w);
    var p2 = (v - w);
    print('${p1.x}, ' + '${p1.y}'); → 6, 9
    print('${p2.x}, ' + '${p2.y}'); → 4, 5
}
```

Как и в предыдущем примере здесь + и - использованы как методы экземпляра с аргументами, представленными экземплярами самого класса. Переменные ***p1*** и ***p2*** также представляют экземпляры класса ***Vector***. Из примера можно видеть, какой синтаксис применяется при использовании директивы ***operator***.

Переменные экземпляра можно объявить со спецификатором ***final*** и тогда нельзя будет изменять их значение в экземпляре класса:

```
class A {
    final String name;
    final t = DateTime.now();
    A(this.name);
    A.pp() : name = '';
}
void main() {
    var p = A('Anna');
    print(p.name); →Anna
    var q = A.pp();
    print(q.t); → 2022-01-16 19:46:54.405600
}
```

В этом случае нельзя выполнить, например, такое присваивание:

```
p.name = 'Marta';
```

К основному конструктору можно добавлять так называемый «досылочный» (Redirecting) конструктор, обычно именованный, позволяющий задавать значения по умолчанию некоторым из параметров в конструкторе:

```
class Point {
  double x, y;
  Point(this.x, this.y);
  Point.aa(double x) : this(x, 9);
}
void main() {
    var p = Point(3, 4);
```

```
    var p1 = Point.aa(7);
    print('(${p1.x}, ${p1.y})'); → (7.0, 9.0)
}
```

Экземпляры класса можно создавать константными. Для этого надо конструктор объявить со спецификатором ***const***, а поля — со спецификатором ***final***:

```
class Point {
    static const Point origin = Point(0, 0);
    final double x, y;
    const Point(this.x, this.y);
}
void main() {
    const p = Point(3, 4);
    print('(${p.x}, ${p.y})'); → (3.0, 4.0)
    print('(${Point.origin.x}, ${Point.origin.y})'); → (0.0, 0.0)
}
```

В этом случае нельзя изменить экземпляр класса, то-есть, нельзя например добавить строку

```
p = Point(7, 9);
```

При использовании спецификатора ***static*** можно создавать переменные класса, их значение вызывается по имени самого класса с добавлением имени статического экземпляра (в данном примере ***origin***).

В конструкторе можно использовать охрану (guard) удобнее всего с использованием оператора ***assert***:

```
class Point {
    double x, y;
    Point(this.x, this.y) : assert(x >= 0);
}
void main() {
    var p = Point(-2,6);
    print('(${p.x}, ${p.y})'); →(2.0, 6.0)
}
```

При отрицательном значении ***x*** будет исключение:

```
var p = Point(-2,6); → Failed assertion line 3 pos 33: 'x >= 0'
```

Напоминаю, что при этом команда запуска программы должна быть с ключом:

dart --enable-asserts prog.dart

Встроенные методы ***get*** и ***set*** позволяют создавать дополнительные поля (свойства) объекта, обеспечивая доступ к ним по чтению и по записи соответственно. Тело этих методов можно задавать самостоятельно (записывается после стрелки => или в фигурных скобках с оператором ***return***), обеспечивая нужную функциональность. В методе ***set*** поле задаётся в виде функции, принимающей аргумент и позволяющей выполнять любые действия, например изменять другие поля объекта. Посмотрим на примере:

```
class A {
        double x, y;
        A(this.x, this.y);
        double get z => x + y;
        set z(double t) => x = t - y;
}
void main() {
        var p = A(3, 20);
        print(p.x); → 3.0
        print(p.z); → 23.0
        p.z = 12;
        print(p.x); → -8.0
}
```

Значит, задавая значение ***p.z = 12;*** мы инициируем этим значением аргумент ***t***, использованный потом для вычисления переменной ***x***.

Методы экземпляра могут быть абстрактными, не имеющими тела, они объявляются обязательно в абстрактном классе. Эти методы могут переопределяться в дочернем классе с учётом ограничений, наложенных на них в родительском абстрактном классе, например:

```
abstract class A {
        double f();
}
class B extends A {
        double f() {return 25.9;}
}
void main() {
        var p = B();
        print(p.f()); → 25.9
```

```
}
```

Здесь в дочернем классе ***B*** метод ***f*** может возвращать только тип ***double***, как это задано в классе ***A***.

Абстрактный класс может иметь также конструктор и поля, но он не позволяет создавать экземпляры класса, это можно делать только в дочерних не абстрактных классах.

Кроме дочерних классов создаваемых с использованием слова ***extends***, можно создавать некие классы двойники применив слово ***implements.*** Экземпляры таких классов можно использовать наравне с основным классом, например:

```
class A {
        final String x;
        A(this.x);
        String f(String y) {return 'Привет, $y. Меня зовут $x.';}
}
class B implements A {
        String get x {return 'aaa';}
        String f(String z) {return 'Меня зовут $z';}
}
String g(A p) {return p.f('Борис');}
void main() {
        print(g(A('Катя')));  → Привет, Борис. Меня зовут Катя.
        print(g(B()));  → Меня зовут Борис
        var t = B(); print(t.x);  → aaa
}
```

Здесь аргумент функции ***g*** представлен экземпляром класса ***A***, но функция способна принимать и экземпляр класса ***B*** тоже. При этом в классе ***B*** переопределен метод **f** из класса ***A***. Доступ в классе ***B*** к полю ***x*** из класса ***A*** можно получить с помощью метода ***get***, при этом в классе ***A*** это поле должно быть с модификатором ***final***.

При наследовании класса директива ***super*** позволяет ссылаться на метод родительского класса, например:

```
class A {
        void f() {
                print('Hello, World!');
        }
}
```

```
class B extends A {
        void g() {
                super.f();
                print('Привет!');
        }
}
void main() {
        var p = B();
        p.g(); → Hello, World! Привет!
}
```

Метод ***g*** дочернего класса ***B*** использует метод ***f*** из родительского класса ***A*** с помощью ***super***.

Dart позволяет использовать миксины (***mixin***) для «подмешивания» дополнительного кода к классу, изменяя его функциональность. Для использования миксина применяется слово ***with***, например:

```
import "dart:math";
class A with B {
        double x;
        A(this.x);
        double f() {return sin(x);}
}
mixin B {
        double g(sn, y) {return sn/cos(y);}
}
void main() {
        var p = A(3.0);
        var tn = p.g(p.f(), 7.0);
        print(tn); → 0.18718608048571422
}
```

Здесь вычисляется тангенс по формуле tn = sin(3)/cos(7). В классе ***A*** определён метод ***f***, вычисляющий синус, а в миксине ***B*** определён метод ***g***, вычисляющий тангенс. Как видим, метод ***g*** доступен в экземпляре класса ***A***.

Иногда бывает полезно ограничить применение миксина каким-то одним классом (типом). Это можно сделать с применением слова ***on***:

```
class A { … }
mixin B on A { … }
```

class C extends A with B { … }

Здесь мы указываем, что миксин ***B*** может применяться только для класса ***A***. Класс ***C*** наследует классу **A** вместе с его миксином ***B***.

Если в классе объявить переменную с модификатором ***static***, то эта переменная будет доступна по имени класса, без создания экземпляра класса. Обычно такие переменные называют переменными или полями класса, можно считать, что они принадлежат самому классу.

```
class A {
        static var x = 5;
}
void main() {
        print(A.x); → 5
        A.x = 7;
        print(A.x); → 7
}
```

Переменная класса доступна как по чтению, так и по записи. Она может использоваться, например, для подсчёта числа создаваемых экземпляров класса:

```
class A {
        static int x = 0;
        A() { x++; }
}
void main() {
        var p1 = A();
        var p2 = A();
        var p3 = A();
        print(A.x); → 3
}
```

Здесь переменная класса ***x*** изменяется в блоке, добавленном к конструктору класса (смотрите ранее).

Аналогично создаются методы класса, которые также доступны по имени самого класса:

```
import 'dart:math';
class Point {
  double x, y;
  Point(this.x, this.y);
  static double f(Point a, Point b) {
```

```
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}
void main() {
  var a = Point(2, 2);
  var b = Point(5, 6);
  var d = Point.f(a, b);
  print(d); → 5.0
}
```

Здесь функция ***f*** представляет метод класса.

7. Параметры типа

генерики (Generics)

Параметры типа отличаются от обычных параметров тем, что им можно присваивать не значения, а типы. Часто для параметров типа используют термин генерики (generic - настраиваемый, родовой, групповой). Применение генериков делает программу более компактной и выразительной, позволяя избегать дублирования кода. Обычно параметры типа обозначают одиночными заглавными буквами, чаще всего T, E, S или K. Синтаксически параметр типа располагается в угловых скобках. Ранее мы уже применяли параметры типа. Например, объявляя и инициируя список ***m***:

```
List<int> m = [1, 2, 3];
```

мы применяли параметр типа с конкретным его значением ***int***.
Посмотрим на более содержательный пример:

```
abstract class A<T> {
        T f();
        List<T> m = [];
}
class B extends A<double> {
        double f() {return 25.9;}
        List<double>m = [1.2, 4.5,2.0];
}
void main() {
```

```
    var p = B();
    print(p.f()); → 25,9
    print(p.m); → [1.2, 4.5, 2.0]
}
```

В абстрактном классе ***A*** введён параметр типа ***T*** (в угловых скобках после идентификатора класса) и объявлен метод ***f***, возвращающий результат типа ***T***, а также объявлен список ***m*** с элементами типа ***T***. При реализации метода в дочернем классе ***B*** параметру типа задаётся конкретное значение ***double.*** При этом можно было бы задать и любое другое значение типа, но задав его мы уже не можем определить метод ***f***, возвращающий результат другого типа, например в данном случае мы не можем определить функцию так:

```
int f() {return 77;}
```

Точно также мы задаём значение параметра типа для списка ***m***.

Таким образом мы получаем возможность один и тот же абстрактный класс использовать для работы с разными типами. Отметим ещё, что явно указывать типы в классе ***B*** не обязательно и допустим такой вариант:

```
class B extends A<double> {
    f() {return 25.9;}
    var m = [1.2, 4.5,2.0];
}
```

Dart позволяет тестировать параметр типа с помощью оператора ***is***, например:

```
var m = <String>[];
m.addAll(['Seth', 'Kathy', 'Lars']);
print(m is List<String>); → true
```

Передавать параметру типа конкретное значение можно в конструкторе класса:

```
class A<T> {
    T? x;
    A(this.x);
}
void main() {
    var p = A<String>('Anna');
    print(p.x); → Anna
}
```

Поскольку невозможно присвоить конкретное значение переменной типа ***T***, то она объявлена nullable, то-есть со знаком вопроса: ***T?***. Конструктор не обязательно должен быть в явном виде:

```
class A<T> {
    T? x;
}
void main() {
    var p = A<String>();
    p.x = 'Anna';
    print(p.x); → Anna
}
```

С помощью оператора наследования ***extends***, например в такой форме <***S extends A***>, можно вводить конкретное ограничение для параметра типа:

```
class A {
    String x = "Hello";
}
class B<S extends A> {
    S y;
    B(this.y);
}
void main() {
    var p1 = A();
    var p2 = B(p1);
    var r = p2.y;
    print(r); → Instance of 'A'
    print(r.x); → Hello
}
```

В классе ***B*** параметр типа ***S*** может принимать только значение ***A*** и это пользовательский тип, созданный классом ***A***. В выражении

```
var p2 = B(p1);
```

мы передаём конструктору класса ***B*** экземпляр класса ***A***, которым затем инициирована переменная ***r***.

Параметр типа может быть аргументом функции, результатом, возвращаемым функцией, значением локальной переменной, например:

```
T f<T>(List<T> m) {
```

```
        T x = m[1];
        return x;
}
void main() {
        var y = f([int, double, String]);
        print(y); → double
}
```

Здесь функция ***f*** принимает список ***m***, элементы которого представлены типами и возвращает тип, которым инициирована переменная ***x***. Параметр типа ***T*** вводится как обычно в угловых скобках после идентификатора функции. Приведу ещё один пример некоторых манипуляций с параметрами типа:

```
class A {
        String x = 'Hello';
}
class B {
        double y = 0;
        B(this.y);
        double f(double z) {
                return(y + z);
        }
}
List<T> g<T>(List<T> m) {
        T t = m[0];
        T s = m[1];
        return [t, s];
}
void main() {
        List p = g([A().x, B(2).f(3.0)]);
        print(p); → [Hello, 5.0]
}
```

Dart позволяет вводить псевдонимы (alias) для обозначения типов с помощью ключевого слова ***typedef***. Применение псевдонимов делает код более кратким и выразительным:

```
typedef IL = List<int>;
void main() {
   IL m = [1, 2, 3];
```

```
   print(m); → [1, 2, 3]
}
```

Здесь для обозначения списка целых чисел введён псевдоним ***IL***. Объявлять псевдоним требуется во внешнем пространстве имён. Псевдоним типа может иметь параметры типа:

```
typedef LM<T> = Map<T, List<T>>;
void main() {
   LM<int> h = {};
   h = { 1: [1,2,3], 2: [4,5,6], 3: [7,8,9] };
   print(h); → { 1: [1,2,3], 2: [4,5,6], 3: [7,8,9] }
}
```

Для хеша со значениями в форме списка принят псевдоним ***LM*** с параметром типа ***T***. При объявлении хеша **h** параметру типа задано конкретное значение ***int***. Без применения псевдонима объявление хеша ***h*** должно бы быть таким:

```
Map<int, List<int>> h = {};
```

С помощью ***typedef*** можно ввести псевдоним типа для функции:

```
typedef F< T > = int Function(T , T ) ;
int f(String a , String b) => a.length + b.length;
void main () {
      print(f is F< String >); → true
      print(f("Людмила", "Борис")); → 12
}
```

С помощью оператора ***is*** убеждаемся, что функция ***f*** соответствует типу ***int Function(T , T)***, обозначенному псевдонимом ***F***.

8. Асинхронные функции

Dart позволяет использовать функции, которые возвращают ***Future*** или ***Stream*** объекты. Такие функции относятся к асинхронным и дают возможность реализовать параллельное выполнение трудоёмких (затратных по времени) операций, например таких, как ввод-вывод или работа с интернетом. После вызова асинхронной функции программа продолжает работу параллельно не дожидаясь окончания трудоёмкой операции. Много асинхронных функций имеется в библиотеках Dart, можно также создавать их самостоятельно. Для того, чтобы функция возвращала результат типа ***Future*** достаточно

применить модификатор ***async***, который располагается перед телом функции, то-есть перед фигурной скобкой или стрелкой =>, если функция задана с применением arrow – синтаксиса. Рассмотрим простой пример:

```
f() async {
        List<int> m = [];
        for (var i = 1; i <= 10; i++) { m.add(i); }
        return m;
}
void main() {
        var x = f();
        print( '$x Конец');→ Instance of 'Future<dynamic>' Конец
}
```

Здесь асинхронная функция ***f*** вычисляет список **m**, но вызывающая эту функцию программа не дождалась результата и вместо ожидаемого списка мы получили текст ***Instance of 'Future<dynamic>'***, сообщающий, что функция ***f*** ожидает возвращения объекта типа ***Future<dynamic>***. Параметр типа ***<dynamic>*** выведен из контекста. На практике рекомендуется возвращаемый функцией тип указывать конкретно и тогда будем иметь:

```
Future<List<int>> f() async {
        List<int> m = [];
        for (var i = 1; i <= 10; i++) { m.add(i); }
        return m;
}
void main() {
        var x = f();
        print( '$x Конец');  → Instance of 'Future<List<int>>' Конец
}
```

В данном примере результат вычисления списка оказался не использованным, хотя все вычисления были выполнены, позже мы это проверим. Асинхронную функцию легко заставить выполняться синхронно, для этого достаточно при вызове функции применить модификатор ***await***:

```
Future<List<int>> f() async {
        List<int> m = [];
        for (var i = 1; i <= 10; i++) { m.add(i); }
```

```
    return m;
}
void main() async {
    var x = await f();
    print( '$x Конец');  → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Конец
}
```

Модификатор ***await*** можно применять только в теле асинхронной функции, поэтому перед телом функции ***main*** также потребовалось поставить слово ***async***. Рекомендуется также для ничего не возвращающих асинхронных функций указывать тип результата в таком виде:

Future<void> main() async {...}

Для того, чтобы убедиться, что в нашем примере асинхронная функция ***f*** вычисляет результат, хотя он и не был нами использован, немного изменим программу применив класс:

```
class A {
    List<int> m = [];
    Future<List<int>> f() async {
        for (var i = 1; i <= 10; i++) { m.add(i); }
        return m;
    }
}
Future<void> main() async {
    var p = A();
    var x = p.f();
    print('$x Конец');  → Instance of 'Future<List<int>>' Конец
    print(p.m);  → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

Таким способом мы смогли вывести вычисленный список ***m*** на терминал. Если применить модификатор ***await*** при вызове метода ***f***:

var x = await p.f();

то получим такой результат:

```
print('$x Конец');  → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Конец
print(p.m);  → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Посмотрим теперь на пример с использованием встроенной асинхронной функции:

```
Future<void> f() async {
```

```
    return Future.delayed(Duration(seconds: 2),
        () => print('Дождались'));
}
void main() {
    f();
    print('Конец');
}
```

Будет выведено две строки:

Конец
Дождались

Здесь в теле функции ***f*** использована функция (метод) ***delayed***, принадлежащая классу Future (delayed – ожидаемый, опаздывающий). Эта функция принимает два аргумента: оператор ***Duration*** (длительность, продолжительность) с параметром ***seconds: 2*** указывающим продолжительность задержки в две секунды, и анонимную функцию, выводящую на терминал слово *Дождались*.

Как видим, программа не дождалась выполнения асинхронной функции ***f*** и вывела на печать слово *Конец*. Однако, функция ***f*** после заданной задержки была выполнена и вывела на терминал слово *Дождались* после слова *Конец*.

Применим теперь модификатор ***await*** при вызове функции ***f***, при этом саму вызывающую функцию ***main*** делаем тоже асинхронной:

```
Future<void> f() async {
    return Future.delayed(Duration(seconds: 2),
        () => print('Дождались'));
}
Future<void> main() async {
    await f();
    print('Конец');
}
```

Результат изменится и порядок вывода слов будет противоположный:

Дождались
Конец

В общем случае работа выполняется так: асинхронная функция (например main) выполняется до тех пор, пока не встретит вызов другой асинхронной функции с модификатором await. Тогда

вызывающая функция останавливается и ожидает окончания работы вызванной функции, после чего основной поток программы продолжается. Таких остановок может быть несколько. Если модификатор await отсутствует остановки не будет, а вычисления будут происходить параллельно.

Функции с модификатором ***async**** возвращают объекты типа ***Stream***, которые представляют события из потока. Посмотрим на такой пример:

```
Future<int> f(Stream<int> m) async {
        var s = 0;
        await for (var t in m) {s += t;}
        return s;
}
Stream<int> g(int n) async* {
        for (int i = 1; i <= n; i++) {yield i;}
}
Future<void> main() async {
        var m = g(10);
        var s = await f(m);
        print(s); → 55
        print(m); → Instance of '_ControllerStream<int>'
}
```

Здесь оператор ***yield*** возвращает событие в виде очередного значения переменной цикла ***i***. Асинхронная функция ***f*** суммирует эти значения. Передаваемый функции ***f*** параметр ***m*** не коллекция, а последовательность событий и имеет тип ***ControllerStream<int>***.

Объекты типа Future и Stream в Dart могут применяться в разных ситуациях и разнообразными способами. Всё это подробно описано в документации Dart, рассматривать все детали здесь вряд ли целесообразно.

Приведу ещё пример использования типа ***Iterable***, который возвращают функции с модификатором ***sync****:

```
Iterable<int> f(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}
void main() {
```

var x = f(5);
for (var i in x) {print(i);}
print(x); → (0, 1, 2, 3, 4)
}

Здесь также использован оператор ***yield***. Обращаю внимание на то, что ***x*** здесь не список, а объект типа ***Iterable<int>***, при выводе его на печать получаем значения элементов в круглых скобках. Такие объекты удобно и выгодно применять в ленивых вычислениях. В документации их называют генераторами. Модификатор ***sync**** создаёт синхронный генератор. Можно применять и асинхронные генераторы при использовании модификатора ***async ****.