Dart 2.X

For FLUTTER

Базовое руководство.

Предисловие:

- Для того, чтобы писать приложения с помощью **Flutter**, необходимо разобраться с языком программирования Dart. Сам по себе язык Dart гораздо проще C#\Java и тем более C++, я бы сказал это один из самых простых статически типизированных языков программирования на сегодняшний день …
В данном руководстве мы максимально быстро разберем базу языка Dart что бы сразу приступить к написанию мобильных (и не только) приложений на платформе **Flutter**.

От автора:

- Настоятельно рекомендую для максимального эффекта усвоения материала придерживаться простой истины:

1) Не стоит заучивать всё что есть по предмету как законодательство вашей страны – это не поможет и затянет время на долгие годы.

2) Необходимо изучить минимальную базу что бы приступить сразу к практике и уже на практике доучивать то, что вам необходимо.

*3) Только практика, много **полезной** практики. Слово «полезной» - это ключевое слово.*

Требования:

- Вы должны обладать минимальным пониманием что такое программа, как она работает, что такое «классы», «методы», «переменные». Без этих знаний вам будет сложно, но вполне постижимо, было бы желание …

Пройдемся кратко по ключевым словам языка Dart и усвоим некоторые его особенности :

~~~~~~

var и **dynamic**

- **var** способ объявить переменную без указания её типа, при компиляции компилятор сам определит необходимый тип данных.

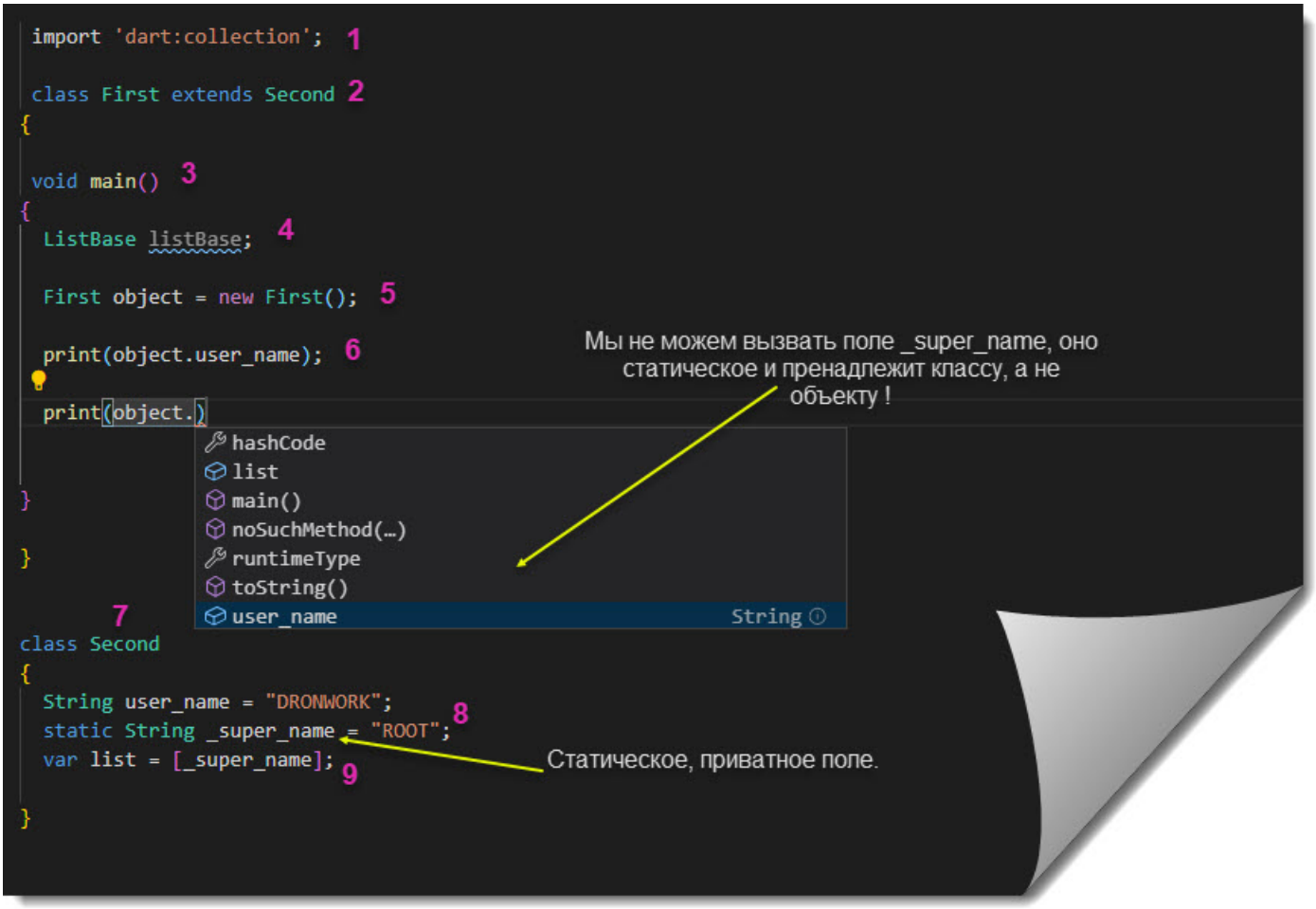
```
void main()
{
  var x = 88; //Тип переменной int
  var str = 'hello'; //Тип переменной string
  var list = [1, 2, 3]; // Тип переменной List
}
```

dynamic оператор, как и var, позволяет вывести тип переменной исходя из присвоенного ей значения. Но в отличие от var, dynamic позволяет изменять тип переменной.

```
void main()
{
  dynamic x = 4;
    print(x); // x = 4
    x =  'Four';     // Тип динамически поменяеться на String
    print(x); // 'Four'
}
```

Константы

- Ключевые слова **final** и **const** используются для объявления констант. Dart предотвращает изменение значений переменной, объявленной с использованием ключевых слов final или const. Эти ключевые слова могут использоваться вместе с типом данных переменной или вместо ключевого слова var.
~~~~~~



Разберем код:

1. Ключевое слово **import** импортирует в нашу программу необходимые библиотеки и классы, что бы использовать их функционал.
2. Новый класс мы создаём с помощью ключевого слова **class** после которого пишем название нашего класса и при необходимости можем наследоваться от другого класса с помощью ключевого слова **extends.** Наследуясь от класса **Second**, мы получаем весь его функционал (поля, методы) за исключением статических членов класса (об этом ниже).
3. Точка входа программы Dart, всегда происходит через метод **main()**, именно с этого метода начинается выполнение программы Dart.
4. **ListBase** – это коллекция из пакета 'dart:collection', так как мы импортировали этот пакет в начале нашей программы, мы можем её использовать.
5. Пример создания объекта класса **First** с конструктором по умолчанию (о конструкторах ещё поговорим).
6. Пример использования объекта.

7. Создание класса **Second**.
8. Статическое и приватное поле **_super_name**. В Dart нет специальных ключевых слов (public; private; protected) то есть спецификаторов доступа, по этому все идентификаторы «публичны» по умолчанию. Вопрос инкапсуляции решен соглашением: приватными считаются все члены класса, чье имя начинается с подчеркивания "_". Кроме обычных полей и методов, класс может иметь статические поля, методы. Статические поля и методы, относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр (объект) класса. Например: **Second._super_name** = “Dell”;
9. Данная конструкция создает *тип* список, который мы рассмотрим ниже.

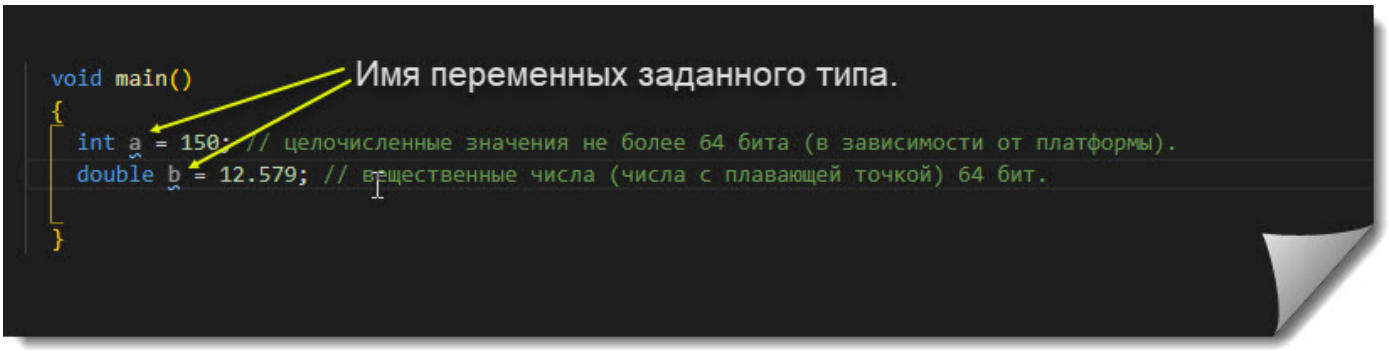
Основы языка **Dart**

Классификация:
1) Объектно-ориентированный.
2) Система типов: **статическая** (с версии 2.0).
3) Расширение: .dart (например файл исходного кода main.dart).
4) Кроссплатформенный.
5) Поддержка JIT и AOT компиляторов, а так же сборщик мусора.

Встроенные типы данных в Dart:

- **numbers** (Числа)
- **strings** (Строки)
- **Booleans** (Логический тип)
- **lists** (Список\Массивы)
- **sets** (Множества)
- **maps** (Словарь\Хэш)
- **runes** (Последовательность символов в кодировке UTF-32)
- **symbols** (Символы)

Numbers (Числа):
- Числовой тип данных двух видов: **int** и **double**



Strings (Строки):

- Строка может быть как одиночной, так и многострочной. Однострочные строки пишутся с использованием одинарных или двойных кавычек, а многострочные строки пишутся с использованием тройных кавычек. Ниже приведены все допустимые строки Dart:

```
void main()
{
  String str = 'Hello';
  String next_str = "Flutter";

  String multiline = """ Многострочная стока
                         Очень много строк
                         так сильно много строк что просто звиздец
                        """;

}
```

Booleans (Логический тип)

- Для работы с логическими значениями в языке Dart есть тип с именем bool. Он предоставляет два значения: **true** и **false**

```
void main()
{

  bool no = false;

  bool yes = true;

}
```

- Более подробно как работать с этим типом данных мы разберем на примере далее в этом руководстве …

List (Список\Массив)

- представляет собой массив с фиксированным или динамическим размером, и с доступом к элементам, осуществляемым при помощи числового индекса. Данная коллекция по определению упорядочена, и соответственно можно полностью управлять тем, куда помещать новые элементы, откуда удалять элементы и сортировать и переупорядочивать существующие. Кроме этого, списки поддерживают операции поиска элементов в коллекции и могут быть обобщенными (параметризированными). Для обращения к элементам массива применяются индексы.

```
void main()
{

 // Создание списка с использованием конструктора List.
 var cars = new List();

// Рекомендуемый способ - использовать литерал List:
var honda = ['Accord', 'Civic', 'CR-V'];

// Добавить в список с помощью метода add:
honda.add('NSX');

// Можно добавить сразу несколько значений:
honda.addAll(['Pilot', 'HR-V']);

// Удалим модель 'Accord' с помощью метода remove
honda.remove('Accord');

// Удалим третий элемент коллекции (будет удален 'CR-V')
honda.removeAt(2);

}
```

Так же тип списка может быть обобщенным (параметризированным), то есть мы можем строго указать тип данных, который будет хранить наша коллекция:

```
void main()
{

 //Тип списка может быть параметризирован
 var subaru = List<String>();
 subaru.add('Forester');
 subaru.add(5); // Возникнет ошибка, список имеет тип String

}
```

Sets (Множества)

- В Dart, коллекция Set представляют собой неупорядоченный набор уникальных элементов. Поскольку элементы не упорядочены, нет возможности получать их по индексу (позиции). Для создания Set применяются фигурные скобки **{}**.

```
void main()
{

    // Создаем коллекцию множеств:
    var sets = {0, 2, 3, 5};

    // Альтернативные возможности создать коллекцию Set:
    Set<int> sets1 = {0, 2, 3, 5};
    var sets2 = <int> {0, 2, 3, 5};
    Set<int> sets3 = <int> {0, 2, 3, 5};
    var ProcessorSet = {
                        'Ryzen',
                        'Core',
                        'Pentium',
                        'Athlon',
                        'Sempron',
};

// Добавляем элементы в нашу коллекцию 'cars' после её создания:
var cars = new Set();
cars.add('Toyota');

// Удаление элемента из коллекции.
cars.remove('Honda');

}
```

Maps (Словарь\хэш)

- известный как словарь (ассоциативный массив) или хэш, это неупорядоченный набор пар ключ-значение. В хеше, для облегчения поиска, значение привязывается к ключу.

Для определения хешей так же необходимы фигурные скобки **{}**, можно использовать простой синтаксис литерала или использовать конструктор.

Стоит отметить что каждый элемент в Map фактически представляет собой объект **Map<K, V>,** где **K** - тип ключей, а **V** - тип значений. У этого типа есть два свойства: key (ключ элемента) и value (значение элемента).


```
void main()
{

    // Создаем хеш с ключами Samsung, Motorola, Xiaomi и их значениями:
var pda = {
  'Samsung'   : ['Note 10', 'Galaxy'],
  'Motorola'  : ['RAZR v3i', 'slv 9'],
  'Xiaomi'    : ['Mi Mix', 'Redmi 8']
};
// Альтернативно:
Map<String, List> pda_next = {
  'Samsung'   : ['Note 10', 'Galaxy'],
  'Motorola'  : ['RAZR v3i', 'slv 9'],
  'Xiaomi'    : ['Mi Mix', 'Redmi 8']
};
// Создание хеш-коллекции где ключ имеет тип int:
var pda_number = {
        1: "Samsung",
        2: "Nokia",
        3: "Meizu"
};
// Альтернативно этому:
Map<int, String> pda_numbers = {
        1: "Samsung",
        2: "Nokia",
        3: "Meizu"
};

// Используя ключи, мы можем получить или изменить значения элементов:
var pda_numbers_next = {
        1: "Samsung",
        2: "Nokia",
        3: "LG"
    };
    pda_numbers[3] = "BlackBerry"; // установим значение BlackBerry вместо LG
    print(pda_numbers[3]);         // BlackBerry
```

Runes (Последовательность символов в кодировке UTF-32)

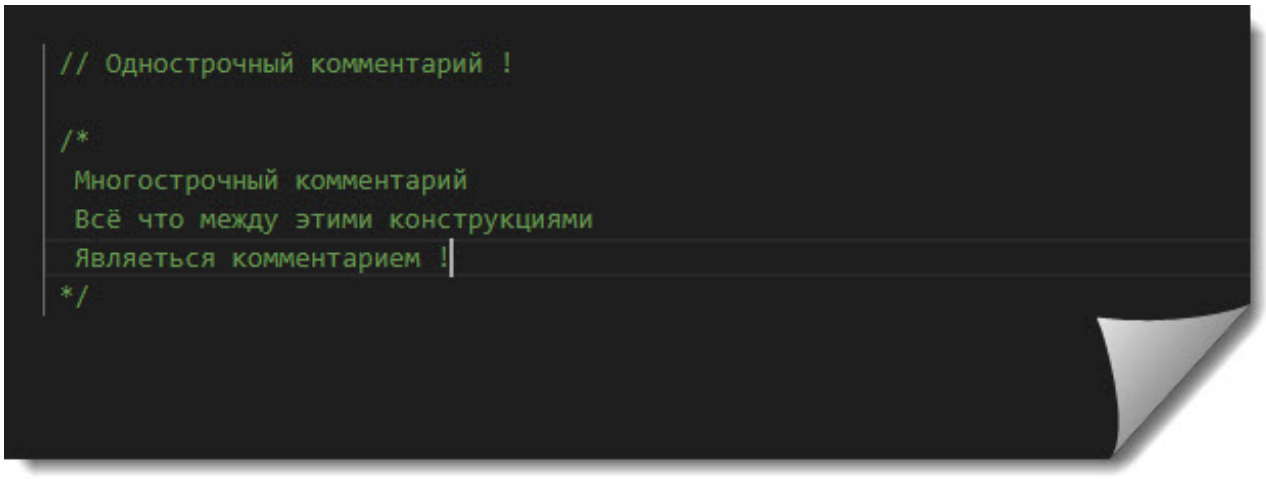
- Данный тип также представляет строки, но в отличие от String, Runes - это последовательность символов в кодировке UTF-32. Поскольку по умолчанию все строки в кавычках (как одинарных, так и двойных) представляют тип String, то для определения переменной Runes требуется специальный синтаксис:

```
void main()
{
    Runes input = new Runes(
        '\u2665  \u{1f605}  \u{1f60e}  \u{1f47b}  \u{1f596}  \u{1f44d}');
  print(new String.fromCharCodes(input));
}
```

Symbols (Символы)
- являются типом Symbol. Для определения объекта этого типа применяется символ решетки #:

```
Symbol sym = #X;  // представляет символ X
```

Комментарии
- В Dart есть два типа комментарий, это однострочный и много строчный:



Прежде чем перейти к более важным частям языка ...

Конструкция if/else

Выражение if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
int a = 5;
int b = 10;

if( b > a ) // Если b больше чем a тогда выполняется условие ниже:
{
    print("b больше чем a");
}
```

```
int a = 5;
int b = 10;

if(b > a)
{
    print("b больше чем a");
}
else
{ // Если выражение ложное, выполняется следующий блок:
    print("a больше чем b");
}
```

После ключевого слова **if** в скобках идет условие.

Если это условие выполняется и возвращает **true**, тогда срабатывает код,

который помещен в блоке **if** после фигурных скобок.

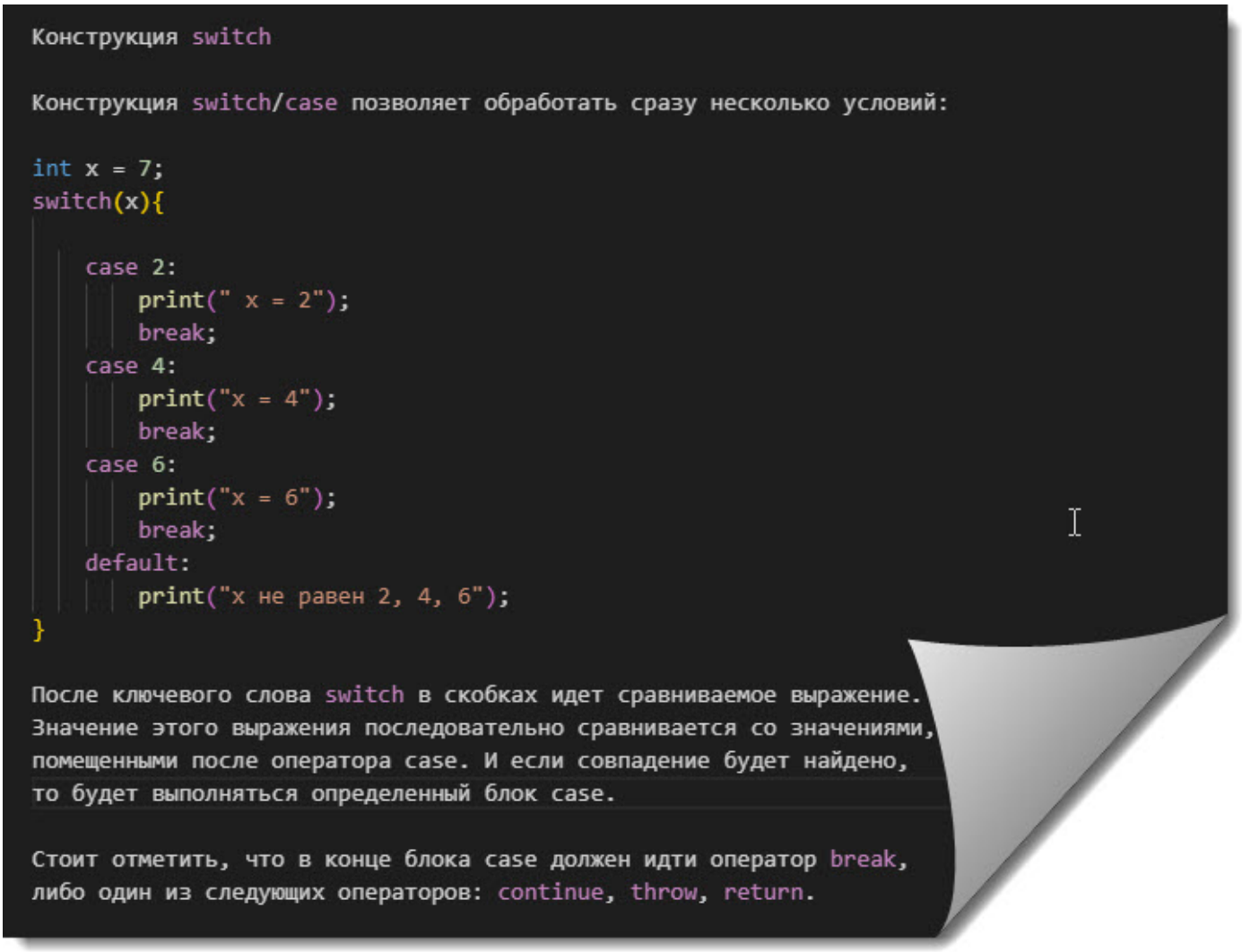
В данном случае в качестве условия выступает операция сравнения двух чисел.

Поскольку здесь **b** больше **a**, то выражение **b > a** истинно и возвращает значение **true**.

Следовательно, управление переходит в блок кода после фигурных скобок и начинает выполнять содержащиеся там инструкции.

Если бы первое число оказалось бы меньше второго или равно ему, то инструкции в блоке **if** не выполнялись бы.

При несоблюдении условия, то есть, когда оно ложное (**false**) могут так же выполнятся действия. В этом случае добавляем блок используя **else**.



- Данные базовые конструкции языка (включая циклы, о которых мы будем говорить далее) особо ничем не отличаются от таковых в других языках (C#\Java\C++) по этому я бы рекомендовал с ними просто ознакомится и пойти дальше, если же вы совсем не знакомы с ними тогда советую прибегнуть к более полному руководству и попрактиковаться. Можно обратится к MSDN там сейчас очень хорошо подается информация …

```
for

while

do...while
```

Цикл for

Цикл for имеет следующее формальное определение:

```
for ([инициализация переменной]; [условие]; [изменение переменной])
{
    // Блок выполнения действия
}
```

Рассмотрим стандартный цикл for:

```
for (int x = 0; x < 5; x++)
{
    print(x);
}
```

Инкремент, аналогично x+1

Первая часть объявления цикла - int x = 0 создает и инициализирует переменную x.
Вторая часть - условие, при котором будет выполняться цикл.
  В данном случае цикл будет выполняться, пока x не достигнет 5.
И третья часть - увеличение переменной на единицу.

В итоге блок цикла сработает 5 раз, пока значение x не станет равным 5.
И каждый раз это значение будет увеличиваться на 1.

Цикл while

Цикл while сразу проверяет истинность некоторого условия,
и если условие истинно, то код цикла выполняется:

```
int a = 10;
while (a > 0)
{
    print(a);
    a--;
}
```

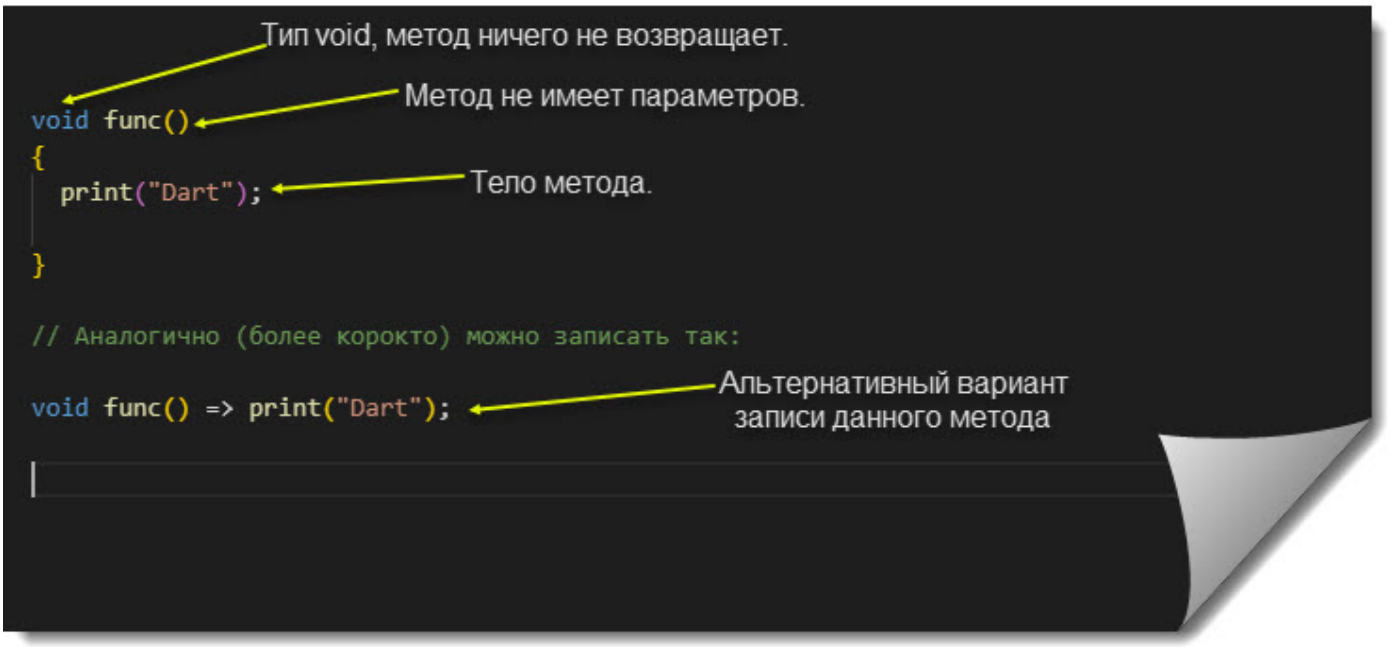
Декремент, аналогично a-1

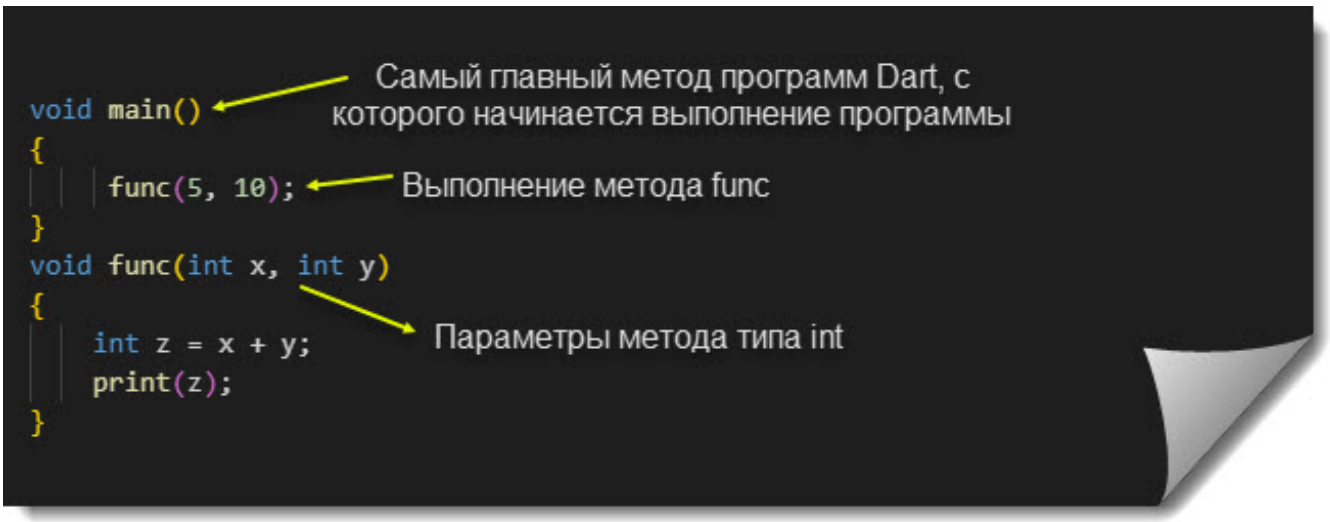
```
Цикл do

Цикл do сначала выполняет код цикла, а потом проверяет условие в инструкции while.
И пока это условие истинно, цикл повторяется. Например:

int z = 4;
do
{
    print(z);
    z--;
}
while (z > 0);
В данном случае код цикла сработает 4 раза, пока z не окажется равным нулю.
Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий,
даже если условие в инструкции while не будет истинно.
```

- *Далее в этом руководстве мы будем разбирать более сложные и необходимые конструкции, часть из которых присущи только языку программирования **Dart**, а именно «Изоляты», «Именованные конструкторы» и прочее.*
- *Начнем от простого к сложному, самая первая и важная задача: как минимум научится читать и понимать код, для **Flutter** есть огромное количество «сэмплов» с исходным кодом, но если вы не сможете прочитать код, тогда особого смысла разбирать как написана эта программа не имеет вовсе …*
- *То есть сейчас главное понять те конструкции, из которых чаще всего состоит программа на **Flutter**.*
- *Ну что же, приступим …*

Методы

- Могут возвращать значение или не возвращать значение, могут иметь тип возвращаемого значения, иметь параметры для выполнения над ними операций в теле метода, рассмотрим некоторые из них:

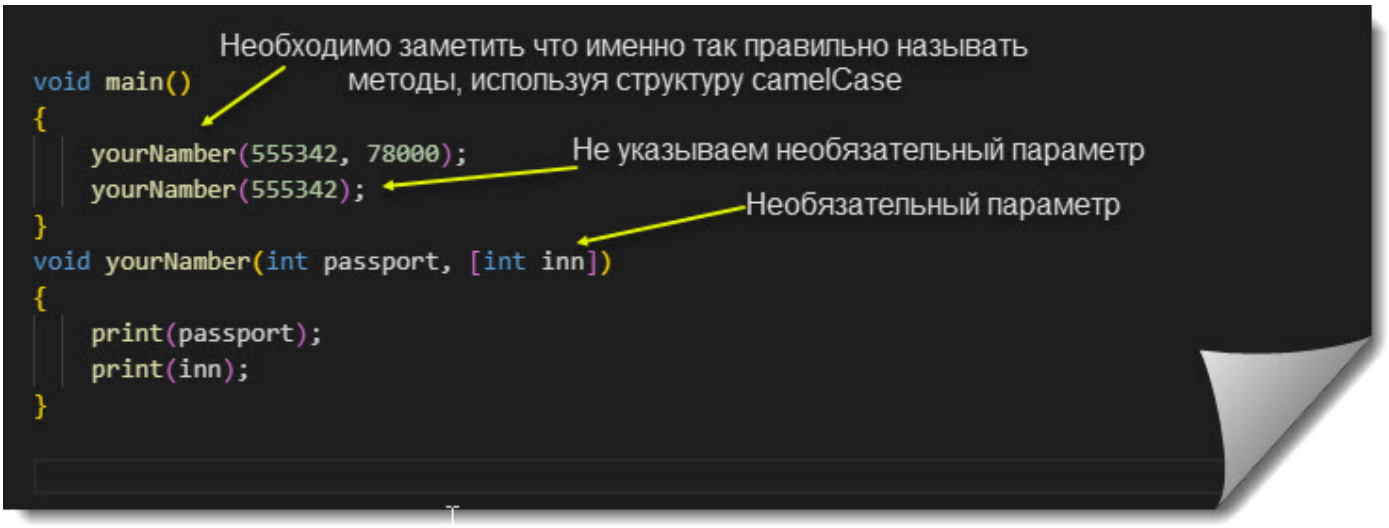




- Эти методы не возвращают значение, не смотря на параметры которые имеет метод мы не можем получить результат сложения, то есть если мы напишем **int result** = **func**(**5**, **10**); - мы получим ошибку так как метод не может вернуть результат сложение данных параметров !

[Важно запомнить]:

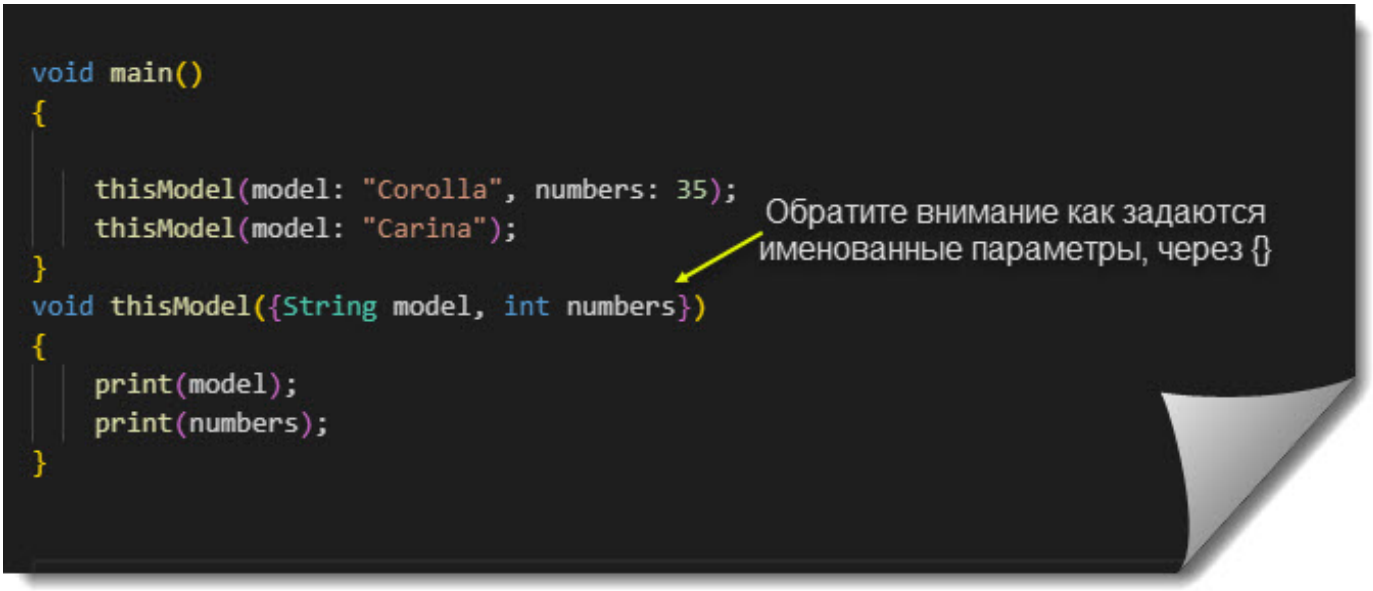
Ряд параметров можно сделать необязательными, то есть мы можем не передавать для них никаких значений. Для этого параметр заключается в квадратные скобки:




```
void main()
{
    yourNamber(555342, 78000);
    yourNamber(555342);
}
void yourNamber(int passport, [int inn])
{
    print(passport);
    print(inn);
}
```

Именованные необязательные параметры (вы часто будете работать с ними используя Flutter).

При вызове функции указывается имя параметра и через двоеточие его значение (**model**: "Corolla").




```
void main()
{

    thisModel(model: "Corolla", numbers: 35);
    thisModel(model: "Carina");
}
void thisModel({String model, int numbers})
{
    print(model);
    print(numbers);
}
```

Параметр, который мы не задали будет иметь тип **null** (ничего).

Так же вы можете определить для параметров значения по умолчанию, если это необходимо:

```dart
void main()
{
    thisModel(); // Результат: Toyota_Model 0
    thisModel(model: "AVALON"); // Результат: AVALON 0
    thisModel(number: 51); // Результат: Toyota_Model 51
}
void thisModel({String model = "Toyota_Model", int number = 0})
{
    print(model);
    print(number);
}
```

- Конечно методы в **Dart** могут возвращать значения того типа который вы укажете при определении метода:

```dart
void main()
{
    int a = func(5, 10);
    print(a);  // результат: 15

    int res = newFunc(10, 5);
    print(res); // результат: 5
}
int func(int x, int y)
{
    return x + y;
}

newFunc(int e, int f)
{
    return e - f;
}
```

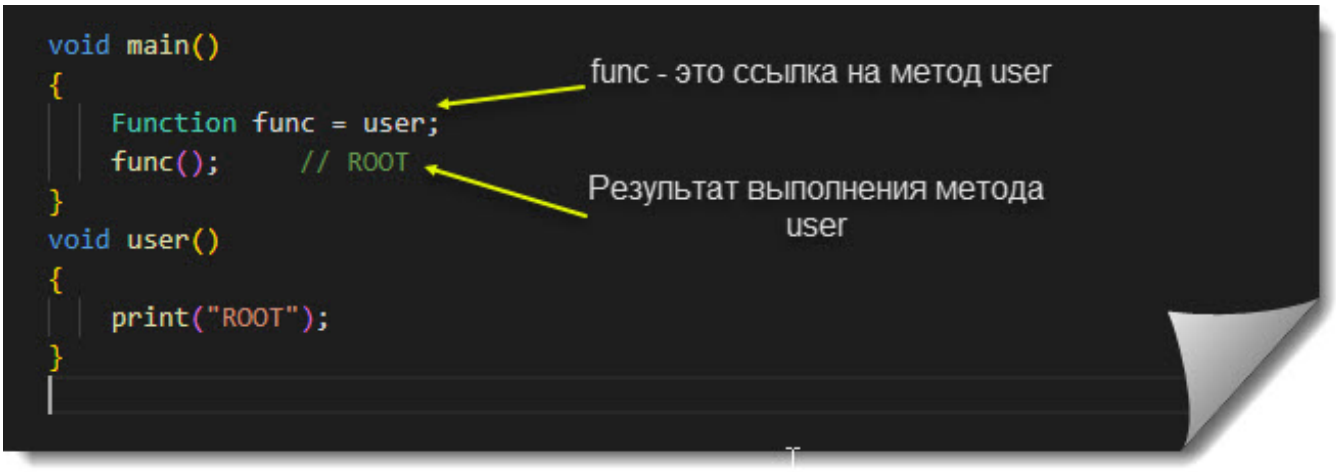
Вызов методов и получение возвращаемого значения

Тип возвращаемого значания

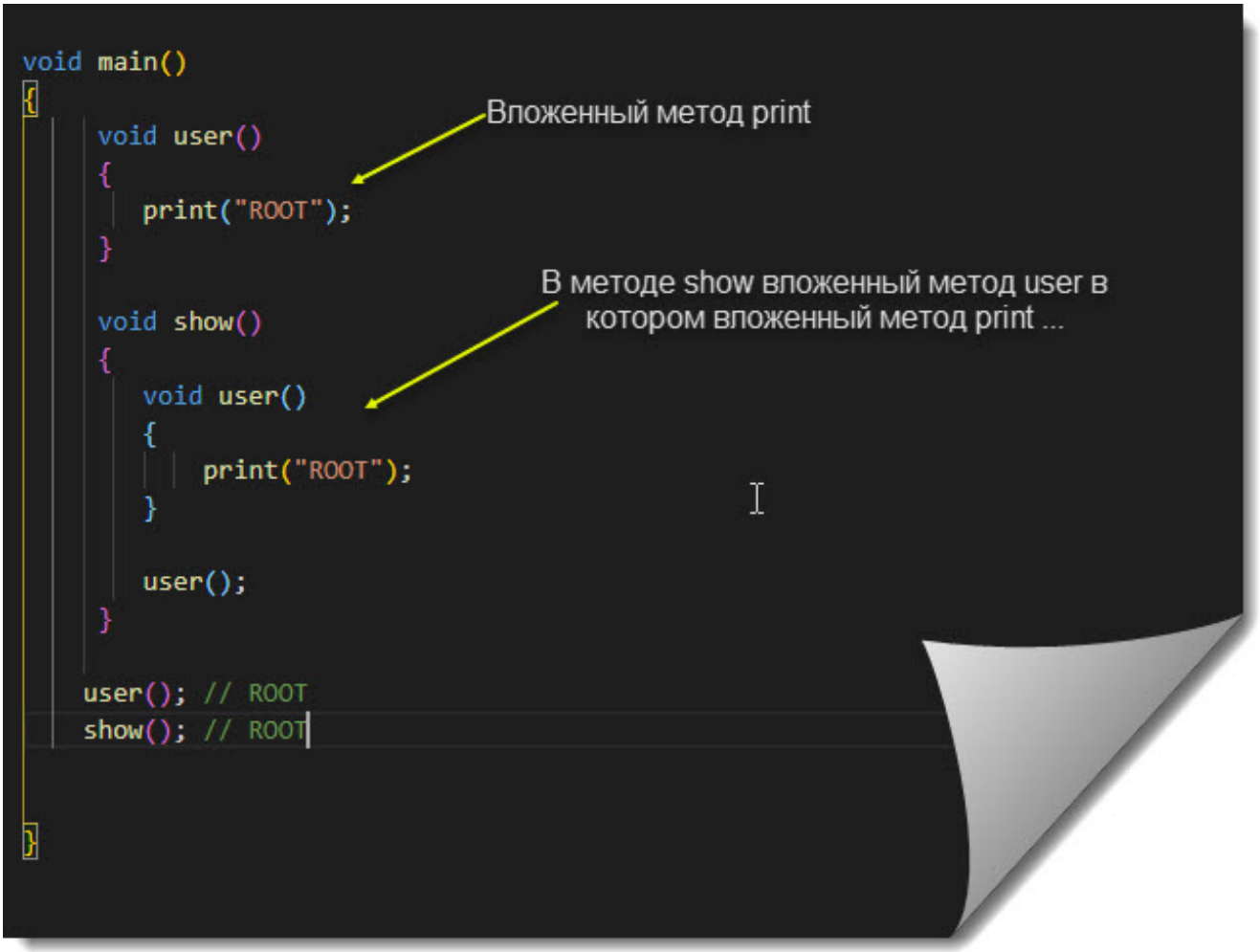
Обратите внимание на данный метод, он так же возвращает значание, но тип этого значения определяет оператор **return**

- Метод может не обязательно может иметь тип возвращаемого значения только **int**, это может быть **String** и даже **dynamic**. Так же в параметрах метода мы можем передавать другой метод …

Методы в Dart – это объекты класса **Function**, то есть наш метод может выступать в качестве отдельного объекта или динамической ссылки на другой метод:



Вложенные методы (мы не однократно это наблюдали на примерах выше)



Классы и объекты в **Dart.**

Класс – это шаблон для вашего объекта, так скажем макет как будет выглядеть и действовать ваш объект, но так же стоит добавить, что **класс** – это тип данных, который вы определяете.

Объект – это представитель класса, имеющий конкретное *состояние* и *поведение* (*состояние* определяют переменные, описанные в классе, а *поведение* методы).

Конструкторы - вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров (конструктор по умолчанию).

[**Конструкторы с параметрами и именованные конструкторы мы разберем далее более подробно**]

```dart
void main () //Метод main, точка входа в программу Dart
{
    ToyotaCars corolla = ToyotaCars(); // Создание объекта corolla класса ToyotaCars
    // Изменяем данные объекта corolla
    corolla.color = "Black";
    corolla.vin_code = 236478;
    // Вызов метода display() (тоесть происходит некое действие объекта corolla)
    corolla.display();
}
 class ToyotaCars
 {
    String color; // Поле класса
    int vin_code; // Ещё одно поле класса
    void display() // Метод, который можно вызвать с помощью объекта данного класса
    {
        print("Color: $color VIN_CODE: $vin_code"); // Знак $ это интерполяция
    }
}
```

Кратко об ООП

Объектно-ориентированное программирование (ООП) — это метод программирования, при использовании которого главными элементами программ являются объекты. В языках программирования понятие объекта реализовано как совокупность свойств (структур данных, характерных для данного объекта), методов их обработки (подпрограмм изменения их свойств) и событий, на которые данный объект может реагировать и, которые приводят, как правило, к изменению свойств объекта. Объединение данных и свойственных им процедур обработки в одном объекте, называется инкапсуляцией и является одним из важнейших принципов ООП.

Другим фундаментальным понятием является **класс**.

Класс - это шаблон, на основе которого может быть создан конкретный программный объект, он описывает свойства и методы, определяющие поведение объектов этого класса. Каждый конкретный объект, имеющий структуру этого класса, называется экземпляром класса.

Следующими важнейшими принципами ООП являются наследование и полиморфизм.

Наследование - предусматривает создание новых классов на базе существующих и позволяет классу потомку иметь (наследовать) свойства класса - родителя.
Полиморфизм - *это способность объекта использовать методы производного класса, который не существует на момент создания базового*. Рожденные объекты обладают информацией о том, какие методы они должны использовать в зависимости от того, в каком месте цепочки они находятся. Если более проще выразится то полиморфизм позволяет переопределять методы.
Инкапсуляция - свойство программ, при котором объекты заключают в себе полное определение их характеристик, никакие определения методов и свойств не должны располагаться вне его, это делает возможным свободное копирование и внедрение одного объекта в другие.

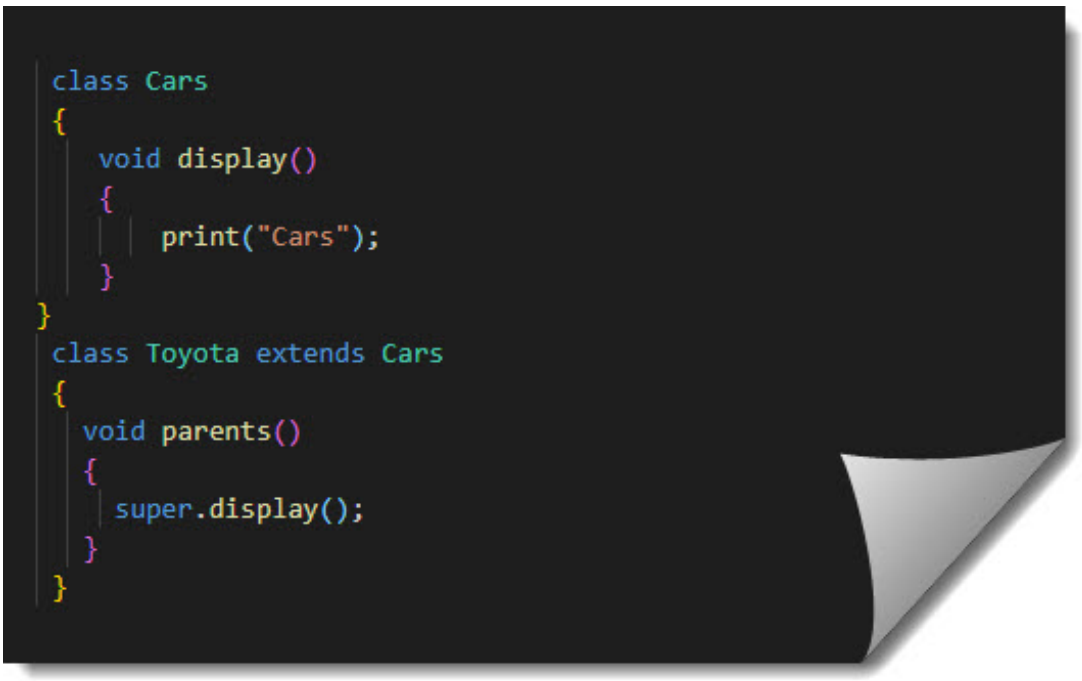
Наследование

- Наследование является одним из фундаментальных атрибутов объектно-ориентированного программирования. Оно позволяет определить дочерний класс, который использует (наследует), расширяет или изменяет возможности родительского класса. Класс, члены которого наследуются, называется базовым классом. Класс, который наследует члены базового класса, называется производным классом.
  Dart поддерживают только одиночное наследование. Это означает, что каждый класс может наследовать члены только одного класса. Но зато поддерживается транзитивное наследование, которое позволяет определить иерархию наследования для набора типов. Другими словами, тип *"Camry"* может наследовать возможности типа *"Toyota"*, который в свою очередь наследует от типа *"Car"*, который наследует от базового класса *"Manufacture"*. Благодаря транзитивности наследования члены типа *"Manufacture"* будут доступны для типа *"Camry"*.

- **Важное дополнение** : *Производные классы не наследуют конструкторы от родительских классов. Производный класс, у которого не объявлены конструкторы имеет только один стандартный конструктор.*
  В языке Dart наследование реализуется с помощью ключевого слова **extends**.

```
void main ()
{
    Toyota carina = Toyota();
    carina.engine_type = "DIESEL";
    carina.display();
}
class Cars{
    String engine_type;
    void display()
    {
        print("Name: $engine_type");
    }
}
class Toyota extends Cars
{
  // Мы можем использовать поле engine_type, с помощью обьекта класса Toyota
}
```

- То есть понятно, что мы имеем доступ к полям и методам класса от которого наследуемся через объект нашего класса.
- Для доступа к функциональности базового класса из производного применяется ключевое слово **super** .



```
class Cars
{
    void display()
    {
        print("Cars");
    }
}
class Toyota extends Cars
{
    void parents()
    {
        super.display();
    }
}
```

Dart не поддерживает множественное наследование как например C++, для того что бы избежать многих проблем связанных с этим, например совпадение имен переменных и методов у предков класса и неоднозначности пути наследования в случае более чем двухуровневой иерархии. Но как альтернатива множественному наследованию в Dart выступают интерфейсы и миксины .

Интерфейсы

- Интерфейсы не содержат какой-либо реализации, но обеспечивают общий тип всем имплементациям (реализациям) всем дочерним классам. Для интерфейсов в Dart нет отдельного ключевого слова, это обычный класс, как правило - абстрактный.

Наследуется интерфейс ключевым словом **implements**.

```dart
// Обычный интерфейс.
class CarsInterface
{
  void car_print() // Метод который должен быть реализован в дочерних классах
}

// Реализация интерфейса для класса Toyota
class Toyota implements CarsInterface
{

  void car_print()
  { // Реализация метода интерфейса
    print("RAV4");
  }
}
```

Мы также можем наследоваться от множества интерфейсов:

```
class Cars implements One, Two, Next
```

Полиморфизм

- Сам термин полиморфизм можно перевести как «много форм». А если говорить простыми словами, полиморфизм – это различная реализация однотипных действий. Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций».
  Как реализуется полиморфизм в Dart :
- Абстрактный метод – это метод, который должен быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое).
- Переопределение метода – это изменение реализации метода, в классе наследнике метод будет работать отлично от базового класса.
  В других языках таких как например C# переопределять можно только виртуальные методы.
  В Dart производные классы могут переопределять (изменять) поведение методов базового класса. Для этого применяется аннотация **@override** :

```dart
void main()
{
  Child obj_c = new Child();
  obj_c.meth(8);
}
class Parent
{
  void meth(int a)
{
print("value of a ${a}");
}
}
class Child extends Parent
{
  @override // Проявление полиморфизма
  void meth(dynamic b)
  { // Изменяем реализацию метода базового класса
    print("value of b ${b}");
  }
}
```

- **Важное дополнение** : число и тип параметров метода должны совпадать при переопределении. В случае несоответствия в количестве параметров или типе данных, компилятор Dart выдаст ошибку.
  Абстрактный метод определяется также, как и обычный, только вместо тела метода после списка параметров идет точка с запятой:
  void nextMethod();

- абстрактные методы могут быть определены только в абстрактных классах. Кроме того, если базовый класс определяет абстрактный метод, то класс-наследник обязательно должен его реализовать, то есть определить тело метода.

```dart
abstract class Magic { // Абстрактный класс

  void what_do_you_want(); // Определение абстрактного метода.
}

class Execution extends Magic {
  void what_do_you_want() {
    print("Хочу Тойоту !!!"); // Реализация абстрактного метода
 }
}
```

Инкапсуляция

Инкапсуляция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные с помощью спецификаторов доступа private, protected, internal - Но в Dart спецификаторов доступа как таковых нет, а приватные поля класса начинается с подчеркивания "_", но **инкапсуляция** реализуется с помощью **геттеров и сеттеров**.

Getters и Setters

- Геттеры и сеттеры — это специальные методы, которые обеспечивают доступ для чтения и записи свойств объекта. Напомним, что каждая переменная класса определяет геттер, и (если это возможно) — сеттер. Вы можете создать дополнительные свойства для реализации геттеров и сеттеров, используя ключевые слова **get** и **set**:

```
class Student
{
  String name;
  int age;

  String get stud_name
{
    return name;
}

  void set stud_name(String name)
{
    this.name = name;
}
}
```

Геттеры и сеттеры применяются во многих языках программирования (C++/C#/Java/Ruby)

Конструкторы

Кроме обычных методов, классы могут определять специальные методы, которые называются **конструкторами**. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.
Разберем строчку кода:

```
ToyotaCars corolla = ToyotaCars();
```

Самое первое мы пишем имя класса **ToyotaCars**, далее имя создаваемого объекта **corolla** и через оператор присвоения (=) метод без параметров **ToyotaCars()** - он же и есть наш конструктор, который выполнить инициализацию нашего объекта **corolla**.

Конструктор с параметрами

Если необходимо что бы при создании объекта поля класса получали какие-то определенные значения тогда используется конструктор с параметрами:

```
void main ()
  {
   ToyotaCars auris = ToyotaCars("Yellow", 234453); //Определяем значения переменных
   auris.display();
  }
class ToyotaCars
{
   String color;
   int vin_code;
   ToyotaCars(String c, int v) //Определяем конструктор с параметрами
   {
       color = c;
       vin_code = v;
   }
   void display()
   {
       print("Color: $color VIN_CODE: $vin_code");
   }
}
```

Ключевое слово **this** - всегда служит ссылкой на объект, для которого был вызван метод. Ключевое слово **this** можно использовать везде, где допускается ссылка на объект типа текущего класса:

```
class ToyotaCars
{
    String color;
    int vin_code;
    ToyotaCars(String color, int vin_code)
    {
        this.color = color;
        this.vin_code = vin_code;
    }

}
```

Либо с помощью this, мы можем сократить определение конструктора:

```
class ToyotaCars
{
    String color;
    int vin_code;
    ToyotaCars(this.color, this.vin_code);
    void display()
    {
        print("Color: $color VIN_CODE: $vin_code");
    }
}
```

Классы могут содержать константные конструкторы.
Такие конструкторы создают объекты, которые не должны изменяться.
Константные конструкторы обозначаются ключевым словом const.
Класс, который определяет подобный конструктор, не должен содержать переменных, но может определять константы.
Кроме того, константные конструкторы не имеют тела:

```
class Ryzen
{
    final String type;
    final String model;
   // константный конструкор
    const Ryzen(this.type, this.model);
}
```

Именованные конструкторы

По умолчанию мы можем определить только один общий конструктор. Если же нам необходимо использовать в классе сразу несколько конструкторов, тогда необходимо применять **именованные конструкторы**, это выглядит как дополнительное имя конструктору, через точку от имени класса.

```
void main ()
{
    ToyotaCars corolla = ToyotaCars.corolla_only(); // Именованный конструктор.
    corolla.display();
    ToyotaCars auris = ToyotaCars.for_auris("Green", 334658); // Второй именованный конст.
    auris.display();
}
 class ToyotaCars
 {
    String color;
    int vin_code;
    ToyotaCars.corolla_only()
    {
        color = "Black";
        vin_code = 784643;
    }
    ToyotaCars.for_auris(String c, int v)
    {
        color = c;
        vin_code = v;
    }
    void display()
    {
        print("Color: $color VIN_CODE: $vin_code");
    }
}
```

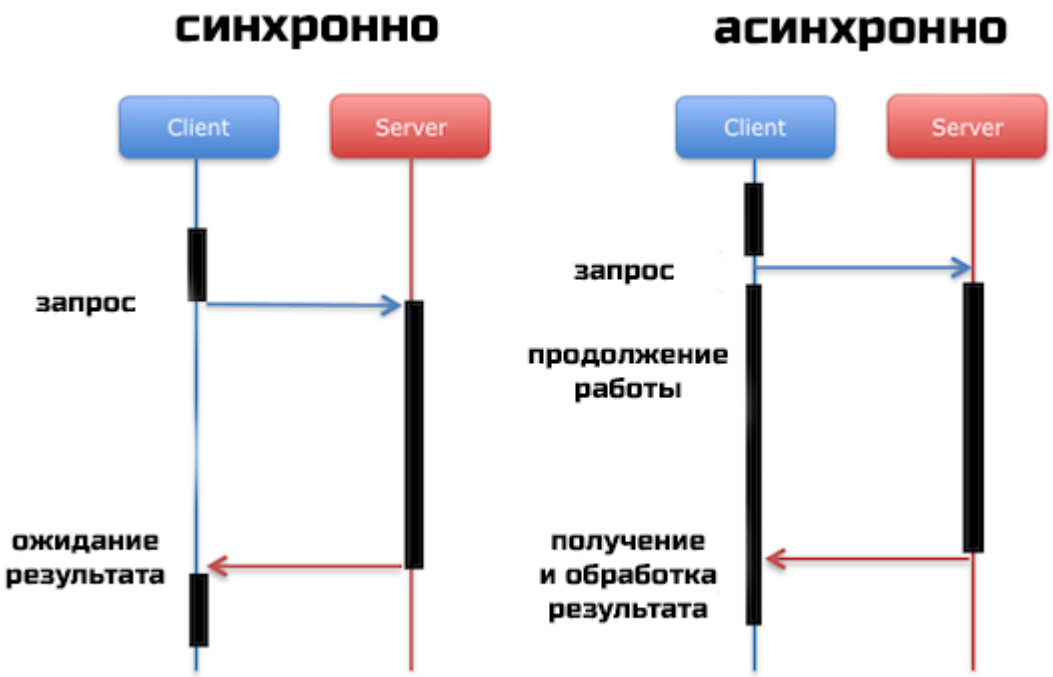
[ПРИМЕЧАНИЕ]:

- Далее вас ждет материал повышенной сложности, который довольно сложно усвоить в теории, сейчас необходимо как минимум с ним просто ознакомится, не более того, на практике (когда будем реализовывать Flutter проекты) многие вещи станут очевидными и понятными …

[асинхронное программирование]

Традиционно в программировании используют **синхронное программирование** — последовательное выполнение инструкций с синхронными системными вызовами, которые полностью блокируют поток выполнения, пока системная операция, например чтение с диска или получение данных из сервера не завершится, до этого вы не будите иметь возможность работать с программой.

- **Асинхронность** (asynchrony) подразумевает, что операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства при этом не блокируя главный поток выполнения программы.
- **Асинхронность в программировании** — выполнение процесса в неблокирующем режиме системного вызова, что позволяет потоку программы продолжить обработку. Проще говоря, главный "процесс" ставит задачу и передает ее другому независимому "процессу", при этом сам продолжает работать.
  Например: Вы решили купить новый аккумулятор на свою "Приору", на сайте или в мобильном приложении в фильтре товаров, выбираете аккумулятор по необходимым вам критериям (Производитель, Ёмкость, Сила тока холодной прокрутки) и после нажатия кнопки "найти товар по заданным критериям" выполняется асинхронная операция поиска на сервере необходимого вам товара, во время этого поиска, вы можете скролить сайт, написать в чат менеджеру магазина, вообщем выполнять ещё какие либо действия пока происходит поиск товара. Если бы операция выполнялась синхронно, вы бы не смогли взаимодействовать с приложением пока не завершится поиск и вывод на страницу товара.



Асинхронное программирование в **Dart**:

Код в Dart работает в одном треде (потоке) выполнения. Если код занят долгими вычислениями или ожидает операцию ввода/вывода, то вся программа приостанавливается.
Dart использует **futures** для представления результатов асинхронных операций. Для работы с **futures** можно использовать *async* и *await* или **Future API**.

- **future** - объект класса Future<T>, который представляет собой асинхронную операцию, возвращающую результат типа **T**. Если результат операции не используются, то тип future указывают Future<void>. При вызове функции, возвращающей future, происходит две вещи:

1. Функция встает в очередь на выполнение и возвращает незавершенный объект **Future**.

2. Когда операция завершена, **future** завершается со значением или ошибкой.

Для написания кода, зависящего от **future**, у вас есть два варианта:

- a) Использовать **async — await**
- b) Использовать Future API

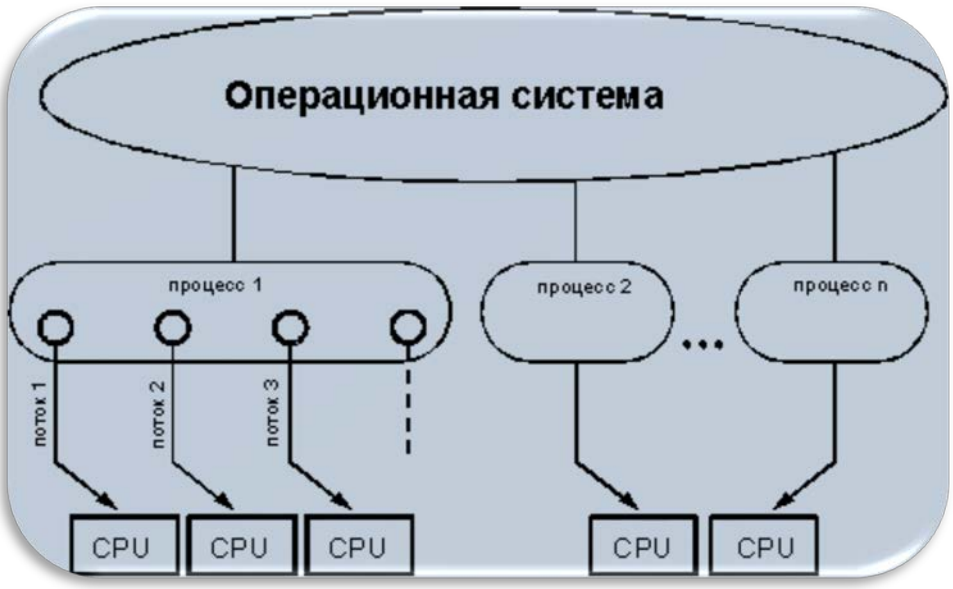
async — await

- Ключевые слова **async** и **await** являются частью поддержки асинхронности в Dart. Они позволяют писать асинхронный код, который выглядит как синхронный код и не использует Future API.
- Асинхронная функция — это функция, перед телом которой находится ключевое слово **async**. Ключевое слово **await** работает только в асинхронных функциях.

Виртуальная машина Dart, Изоляты и Потоки

Dart virtual machine

- Прежде чем говорить о DartVM и остальных вещах в данной теме, необходимо понимать что такое процесс операционной системы и как это всё работает, ибо без этих знаний будет сложно разобраться о чем идет речь далее ...
  Процесс - это (если коротко) выполняющаяся программа либо ваша программа во время выполнения. Программа - это статический обьект представляющий собой файл (или множество файлов) кодом и данными. Для того что бы данную программу можно было запустить (выполнить), ОС должна для данной программы (набор файлов) создать определенное окружение (или среду выполнения), включающую возможность доступа к различным системным ресурсам (память, устройства ввода\вывода и прочее) - такое окружение или среда выполнения получило название **Процесс**.
  Поток - является последовательностью команд, обрабатываемых процессором. В рамках одного **процесса** могут находиться один или несколько **потоков**. Все потоки одного процесса разделяют между собой ресурсы процесса выделенные операционной системой. Они (потоки) находятся в общем адресном пространстве и имеют доступ к одним и тем же данным. Например, если один поток открывает файл для чтения, другие потоки могут читать файл.



VM (понятие виртуальной машины) - это программа, которая действует как компьютер. Она имитирует процессор с несколькими другими аппаратными компонентами, позволяя выполнять арифметику, считывать из памяти и записывать туда, а также взаимодействовать с устройствами ввода-вывода, словно настоящий физический компьютер.
Сколько аппаратного обеспечения имитирует конкретная VM — зависит от её предназначения. Некоторые VM воспроизводят поведение одного конкретного компьютера. создаёт одну стандартную архитектуру CPU, которая симулируется на различных аппаратных устройствах. В первую очередь это делается для облегчения разработки ПО. Представьте, что вы хотите создать программу, работающую на нескольких компьютерных архитектурах. Виртуальная машина даёт стандартную платформу, которая обеспечивает переносимость. Не нужно переписывать программу для разных платформ или ОС. Достаточно сделать только VM которая будет интерпретировать вашу программу для каждой платформы. После этого любую программу можно написать лишь единожды.
Dart_VM - состоит из набора компонентов для выполнения Dart кода. в частности, она включает в себя следующие компоненты:
- Среда исполнения
- "Сборщик мусора"
- Основные библиотеки и нативные методы
- Система отладка
- Профилировщик
- Симулятор ARM архитектуры
- Возможность использовать горячую перезагрузку (с применением JIT-компиляции) и компиляцию AOT (процесс компиляции выполняется полностью перед выполнением программы).

Разработчики Dart VM утверждают что VM которая используется для Dart это скорее среда выполнения (аналогию можно провести с CLR в .NET), нежели виртуальная машина, так как её работа и функциональность далеко за гранью обычно виртуальной машины.

Важно понимать:
Перед запуском вашей Dart программы в ОС, изначально запускается виртуальная машина Dart (или среда выполнения) и внутри её выполняется ваша программа.

Isotales (изоляты)

- Любой код Dart внутри виртуальной машины выполняется в некотором **изоляте**, который можно описать как некий (собственный) процесс Dart VM, так же со своей собственной памятью (кучей) и, как правило, со своим собственным потоком управления. Изолятов может быть большое количество, выполняющих ваш код одновременно, но общаться они могут только через сообщения.
Отношения между потоками ОС и изолятами немного размыты, гарантируется только следующее:

- **Поток ОС может исполнять (принимать) только один изолят за один раз.** Он (поток ОС) должен оставить текущий изолят, если он хочет выполнить другой изолят;
Однако тот же поток ОС может сначала вывести один изолят, выполнить код, затем оставить этот изолят и вывести другой изолят для выполнения.

Использовать изоляты - это единственный способ работы с многопоточностью в Dart.
Приведем пример:
Изоляты могут общаться между собой с помощью сообщений, каждый изолят предоставляет порт, который используется чтобы передать сообщение, этот порт называется **SendPort**.

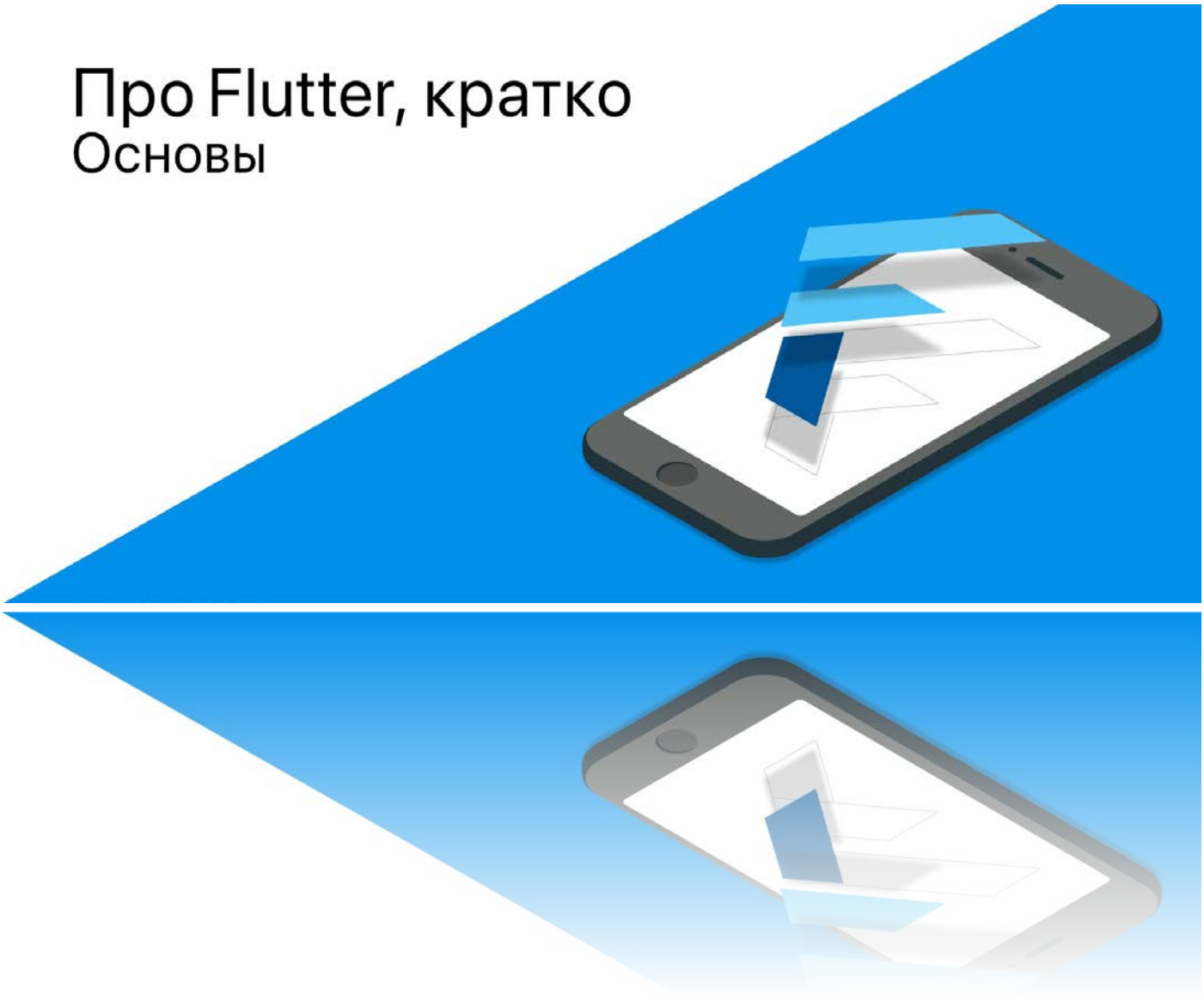
Streams (потоки)

Stream в Dart - это последовательность асинхронных событий. **Stream** сообщает вам, что есть событие и когда оно будет готово.
Существует два типа потоков:
- Потоки-подписки (single subscription)
- Широковещательные (broadcast)
Потоки-подписки - это тип потока который содержит последовательность событий, которые являются частями большего целого. События должны быть доставлены в правильном порядке без пропуска любого из них. Это тип потока, который вы получаете при чтении файла или получении веб-запроса. **Такой поток можно слушать только один раз**. Прослушивание позже, может означать пропуск начальных событий, и тогда остальная

часть потока не имеет смысла. Когда вы начнете слушать, данные будут извлечены и предоставлены кусками.
Широковещательные потоки - тип потока предназначен для отдельных сообщений, которые могут обрабатываться по одному. Вы можете начать слушать такой поток в любое время, и вы получите события, произошедшие во время прослушивания. Поток могут слушать несколько слушателей. Вы можете снова начать слушать события потока после отмены предыдущей подписки.
Знания о **Streams** очень необходимы при работе с **Rx**, сокращенно от **Reactive Extensions** (реактивные расширения), — это потоки "на стероидах". Это концепция, очень похожая на Streams, которая была изобретена для .Net framework командой Microsoft. Так как .Net уже имел тип Stream, который используется для файлового ввода-вывода, они назвали Rx-потоки Observables и создали множество функций для манипулирования данными, проходящими через них. Dart имеет Streams, встроенные в его языковую спецификацию, которые уже предлагают большую часть этой функциональности, но не все. Вот почему был разработан пакет RxDart; он основан на Dart Streams, но расширяет их функциональность.

FLUTTER

BASE



- Забегая на перед, скажу что данное руководство не подразумевает подробное изложение объемного материала по фреймворку Flutter, но некоторые базовые основы может вам дать, вы как минимум сможете понять с чем и как будете работать …

[ПРИМЕЧАНИЕ]:

На конец 2019-го года доступна стабильная версия Flutter 1.9, я готовил материал в конце лета когда была актуальная версия 1.7, но ничего очень важного для нас сейчас там нет, по этому можно смело приступать !

Мобильные приложения и ОС

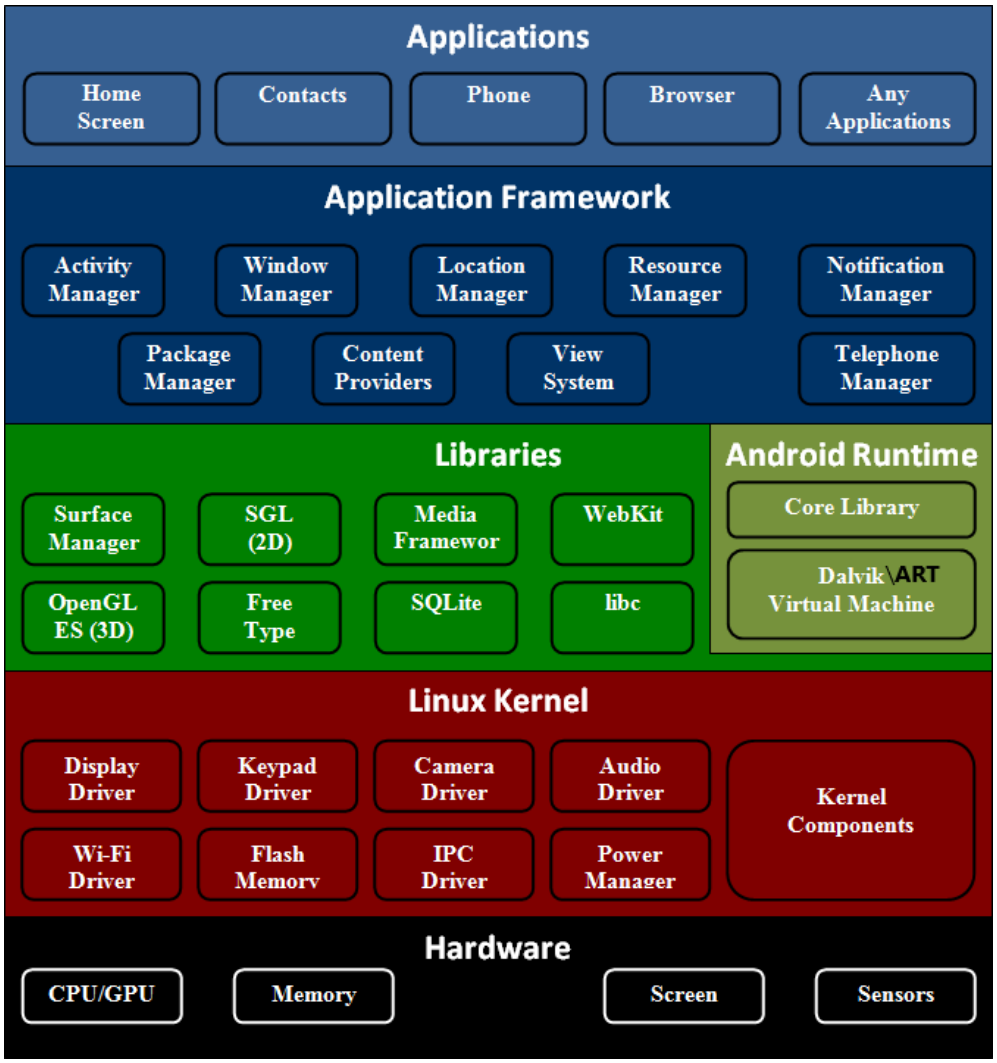
Google создавая такой инструмент как Flutter утверждает что вам не обязательно знать и тем более подробно изучать Android SDK и iOS ибо Flutter – это друг разработчика и он многие нюансы связаны с «нативной» частью берет на себя, но мы думаем что хотя бы ознакомится с некоторыми понятиями стоит.

Сразу оговорюсь что я имел опыт с разработки приложений под Android, по этому здесь чаще будет упоминаться именно эта ОС, но и про «фруктовый» девайс здесь будет сказано моими «яблочными» коллегами.

--- В данной книге (раздел общего положения 2-4) используются материалы из открытых источников включая документацию Google и прочее, ссылки будут указаны в конце данной книги либо в конце текста ---

~ Раздел посвящённый Flutter, особенно практическая его часть основана исключительно на личном опыте и не утверждает что это единственный и правильный подход к разработке мобильного ПО ~

ANDROID OS



Архитектура Android состоит из набора компонентов. Каждый компонент построен на основе элементов более низкого уровня, на рисунке вверху видны самые важные из них.

Ядро

В Android используется модифицированное ядро Linux, которое предоставляет основные драйверы для аппаратных компонентов системы. Кроме того, ядро отвечает за память, управление процессами, поддержку сети и прочее.

Библиотеки

Над уровнем ядра находится уровень инфраструктуры приложения, содержащий виртуальную машину Dalvik/ART, веб-браузер, базу данных SQLite, Java API и набор библиотек.

Набор библиотек написаных на C/C++, создающих основу для фреймворка приложения. Эти библиотеки отвечают за сложные в вычислительном смысле задачи (прорисовка графики, воспроизведение звука, доступ к базе данных), не очень подходящие для виртуальной машины Dalvik\ART (к ним мы ещё вернемся). API в них обернуты с помощью классов Java во фреймворк приложения.

Skia Graphics Library (Skia) – это движок программный визуализатор 20-графики используется для рендеринга пользовательского интерфейса приложений Android (используется так же во Flutter).

OpenGL for Embedded Systems (OpenGL ES), Vulkan – стандарт для аппаратной прорисовки графики.

OpenCore – библиотека записи и воспроизведения аудио- и видеофайлов. Она поддерживает хороший набор форматов (Ogg Vorbis, MP3, H.264, MPEG-4 и т. д

FreeType – библиотека загрузки и обработки растровых и векторных шрифтов (в большинстве случаев формата TrueType). FreeType поддерживает стандарт Юникод, включая написание справа налево для арабских шрифтов и другие подобные случаи.

Framework

Фреймворк приложения связывает вместе системные библиотеки и среду выполнения, создавая таким образом пользовательскую сторону Android. Фреймворк управляет приложениями и предлагает продуманную среду, в которой они работают. Разработчики создают приложения для этого фреймворка с помощью набора программных интерфейсов на Java/Kotlin, охватывающих такие области, как разработка пользовательского интерфейса, фоновые службы, оповещения, управление ресурсами, доступ к периферии и прочее. Все ключевые приложения, поставляемые вместе с ОС Android (например, почтовый клиент), написаны с помощью этих API.

Приложения, будь они с интерфейсом или с фоновыми службами, могут связываться с другими приложениями. Эта связь позволяет одному приложению использовать компоненты других. Простой пример – программа, делающая фотоснимок и потом обрабатывающая его. Приложение запрашивает у системы компонент другого приложения, обеспечивающий это действие. Далее первое приложение может повторно использовать этот компонент (например, от встроенного приложения камеры или от фотогалереи). Подобный алгоритм снимает значительную часть ноши с программиста, а также позволяет настроить многообразие аспектов поведения Android.

Виртуальная машина

Виртуальная машина (VM, от англ. virtual machine) — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (target — целевая, или гостевая платформа) и исполняющая программы для target-платформы на host-платформе (host — хост-платформа, платформа-хозяин) или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы (см.: песочница); также спецификация некоторой вычислительной среды (например: «виртуальная машина языка программирования Си»).

Виртуальная машина исполняет некоторый машинно-независимый код (например, байт-код, шитый код, p-код) или машинный код реального процессора. Помимо процессора, ВМ может эмулировать работу как отдельных компонентов аппаратного обеспечения, так и целого реального компьютера (включая BIOS, оперативную память, жёсткий диск и другие периферийные устройства). В последнем случае в ВМ, как и на реальный компьютер, можно устанавливать операционные системы(например, Windows можно запускать в виртуальной машине под Linux или наоборот). На одном компьютере может функционировать несколько виртуальных машин (это может использоваться для имитации нескольких серверов на одном реальном сервере с целью оптимизации использования ресурсов сервера).

(ВИКИПЕДИЯ)

Dalvik VM

Dalvik — регистровая виртуальная машина для выполнения программ, написанных на языке программирования Java, созданная группой разработчиков Google во главе с Дэном Борнштейном (англ. *Dan Bornstein*). Входит в мобильную операционную систему Android.

Dalvik оптимизирован для низкого потребления памяти, это нестандартная регистр-ориентированная виртуальная машина, хорошо подходящая для исполнения на процессорах RISC-архитектур, часто используемых в мобильных и встраиваемых устройствах, таких как коммуникаторы и планшетные компьютеры (большинство виртуальных машин, используемых в настольных системах, является стек-

ориентированным, включая стандартную виртуальную машину Java, принадлежащую Oracle).

Программы для Dalvik пишутся на языке Java. Несмотря на это, стандартный байт-код Java не используется, вместо него Dalvik исполняет байт-код собственного формата. После компиляции исходных текстов программы на Java (при помощи javac) утилита dx из Android SDK преобразует файлы классов (расширение .class) в файлы собственного формата (с расширением .dex), которые и включаются в пакет приложения (.apk).

В версиях, начиная с Android 4.4 Kitkat, имеется возможность переключиться с Dalvik на более быстрый ART (Android Runtime). В Android 5.0 Dalvik был полностью заменён на ART.

(ВИКИПЕДИЯ)

Android Runtime (ART) VM

Android Runtime — среда выполнения Android-приложений, разработанная компанией Google как замена Dalvik. ART впервые появился в Android 4.4 как тестовая функция, а в Android 5.0 полностью заменил Dalvik. В отличие от Dalvik, который использует JIT-компиляцию (во время выполнения приложения), ART компилирует[1] приложение во время его установки. За счет этого планируется повышение скорости работы программ и одновременно увеличение времени работы от батареи. Недостатком является более долгая загрузка устройства.

Android 7.0 Nougat представила JIT-компилятор с профилированием кода для ART, который позволяет постоянно повышать производительность приложений Android при их запуске. Компилятор JIT дополняет нынешний компилятор Ahead of Time от ART и помогает улучшить производительность во время выполнения.
Для обеспечения обратной совместимости ART использует тот же байт-код, что и Dalvik.

(ВИКИПЕДИЯ)

Android SDK

Пакет средств разработки программного обеспечения для Android (SDK) содержит всеобъемлющий набор инструментов разработки.[Источник 1] Они включают в себя отладчик, библиотеки, эмулятор, основанный на QEMU, документацию, примеры кода и учебники. Поддерживаемые платформы разработки включают компьютеры под управлением Linux (любой современный настольный дистрибутив Linux), Mac OSX и Windows.

Android Studio, созданная компанией Google и работающая над IntelliJ, является официальной IDE; тем не менее, разработчики могут свободно использовать другие инструменты. Кроме того, разработчики могут использовать любой текстовый редактор для редактирования Java и XML файлов, а затем использовать инструменты командной строки (Java Development Kit и Apache Ant не обязательны) для создания, построения и отладки приложений для Android, а также контролировать добавленные Android устройства (например, запуск, перезагрузку, установку пакетов программ удаленно).

Улучшение Android's SDK идут рука об руку с общим развитием платформы Android. SDK также поддерживает старые версии Android, если разработчики хотят сосредоточить свои приложения на старых устройствах. Инструменты разработки - это загружаемые компоненты, поэтому после скачивания последней версии и платформы, старые платформы и инструменты могут быть использованы для тестирования совместимости.[Источник 2]

Приложения Android упакованы в .формат apk и хранятся в папке /Data/App на ОС Android (папка доступна только для пользователя root по соображениям безопасности).

APK -пакет содержит .DEX-файлы[Источник 3] (скомпилированный исполняемый код), файлы ресурсов и прочее.

- О чем ещё необходимо знать ?

1) Файл манифеста **AndroidManifest.xml** предоставляет основную информацию о программе системе. Каждое приложение должно иметь свой файл AndroidManifest.xml.

Назначения файла: объявляет имя Java-пакета приложения, который служит уникальным идентификатором; описывает компоненты приложения — деятельности, службы, приемники широковещательных намерений и контент-провайдеры, что позволяет вызывать классы, которые реализуют каждый из компонентов, и объявляет их намерения; содержит список необходимых разрешений для обращения к защищенным частям API и взаимодействия с другими приложениями; объявляет разрешения, которые сторонние приложения обязаны иметь для взаимодействия с компонентами данного приложения; объявляет минимальный уровень API Android, необходимый для работы приложения; перечисляет связанные библиотеки;

2) **Gradle** — система автоматической сборки, построенная на принципах Apache Ant и Apache Maven. Файл build.gradle, который относится к модулю. Рядом с этим файлом в скобках будет написано Module: app. Также есть файл build.gradle, который относится к проекту.

***AndroidManifest.xml** и **Gradle** – мы будем видеть во Flutter, хотя пользоваться этим будем довольно редко.*

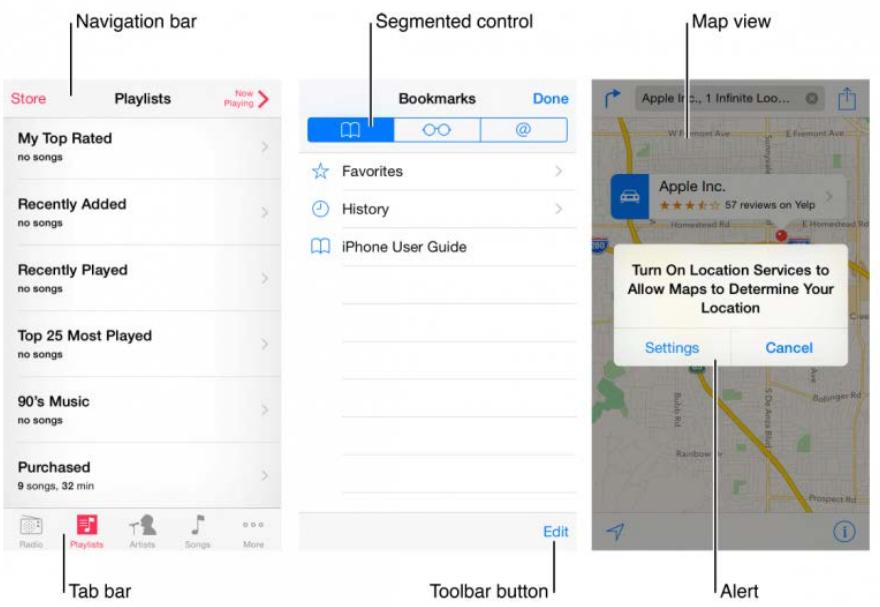
APPLE iOS

iOS (до 24 июня 2010 года — **iPhone OS**) — мобильная операционная система для смартфонов, электронных планшетов, носимых проигрывателей и некоторых других устройств, разрабатываемая и выпускаемая американской компанией Apple. Была выпущена в 2007 году; первоначально — для iPhone и iPod touch, позже — для таких устройств, как iPad . В 2014 году появилась поддержка автомобильных мультимедийных систем Apple CarPlay. В отличие от Android (Google), выпускается только для устройств, производимых фирмой Apple.

В iOS используется ядро XNU, основанное на микроядре Mach и содержащее программный код, разработанный компанией Apple, а также код из ОС NeXTSTEP и FreeBSD. Ядро iOS почти идентично ядру настольной операционной системы Apple macOS (ранее называвшейся OS X). Начиная с самой первой версии, iOS работает только на планшетных компьютерах и смартфонах с процессорами архитектуры ARM.

(ВИКИПЕДИЯ)

Структура iOS приложений



Практически все приложения iOS используют некоторые компоненты пользовательского интерфейса в соответствии с платформой UIKit (***UIKit***Framework (основывается на Application Kit) — библиотека, содержащая специфические для iOS GUI-классы). Знание названий, ролей и способностей этих основных компонентов поможет вам принимать решения в создании пользовательского интерфейса вашего приложения.

Элементы пользовательского интерфейса UIKit подразделяются на 4 категории:

- **Панели.** Панели содержат контекстуальную информацию, которая сообщает пользователям, где они находятся, и управляет для помощи пользователям совершать действия.
- **Обзор содержания.** Обзор содержания включает в себя конкретное содержимое приложения и делает возможными такие операции как, например, прокрутка, вставка, удаление и перестановка иконок.
- **Элементы управления.** Элементы управления выполняют операции или отображают информацию.
- **Всплывающие окна.** Всплывающие окна появляются для предоставления пользователям важной информации или дополнительного выбора и функциональности.

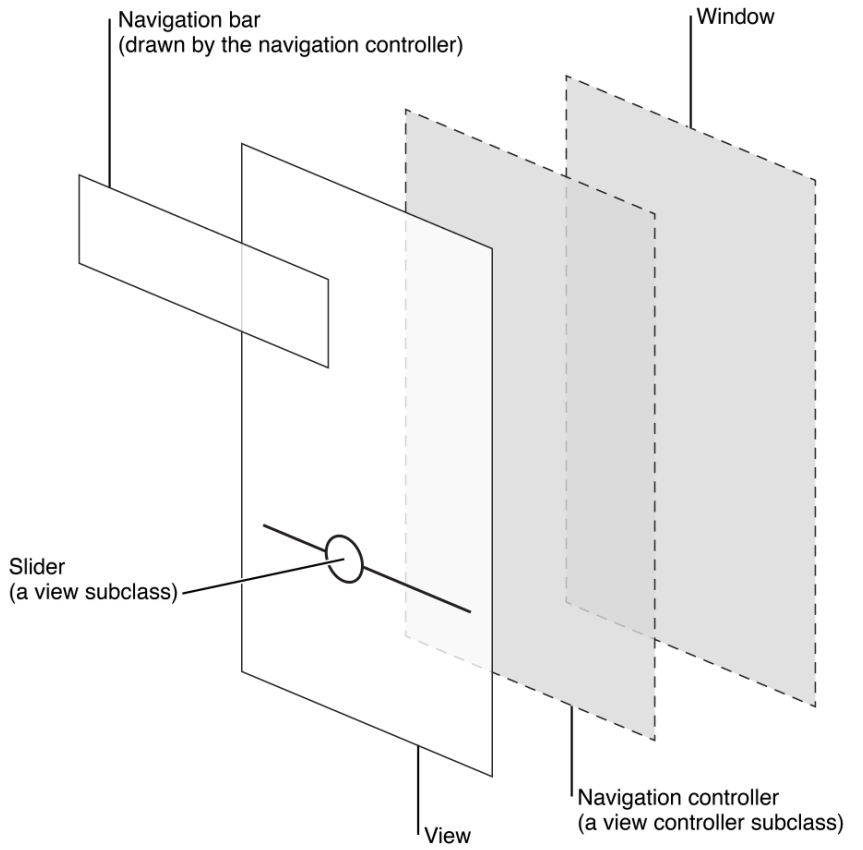
Вдобавок к определению элементов пользовательского интерфейса, UIKit определяет задачи, которые обеспечивают функциональными возможностями, такими, как распознавание жестов, рисование, доступность и поддержка печати.

Программно, элемент пользовательского интерфейса является типом *видов*, так как перенимает признаки от UIView. Вид знает, как отобразить себя на экране, и знает,

когда пользователь дотрагивается в его пределах. Элементы управления (такие, как кнопки и слайды), обзор содержимого (такие, как экранные таблицы и представление коллекций) и всплывающие окна (такие, как предупреждения и листы действий) являются типами видов.

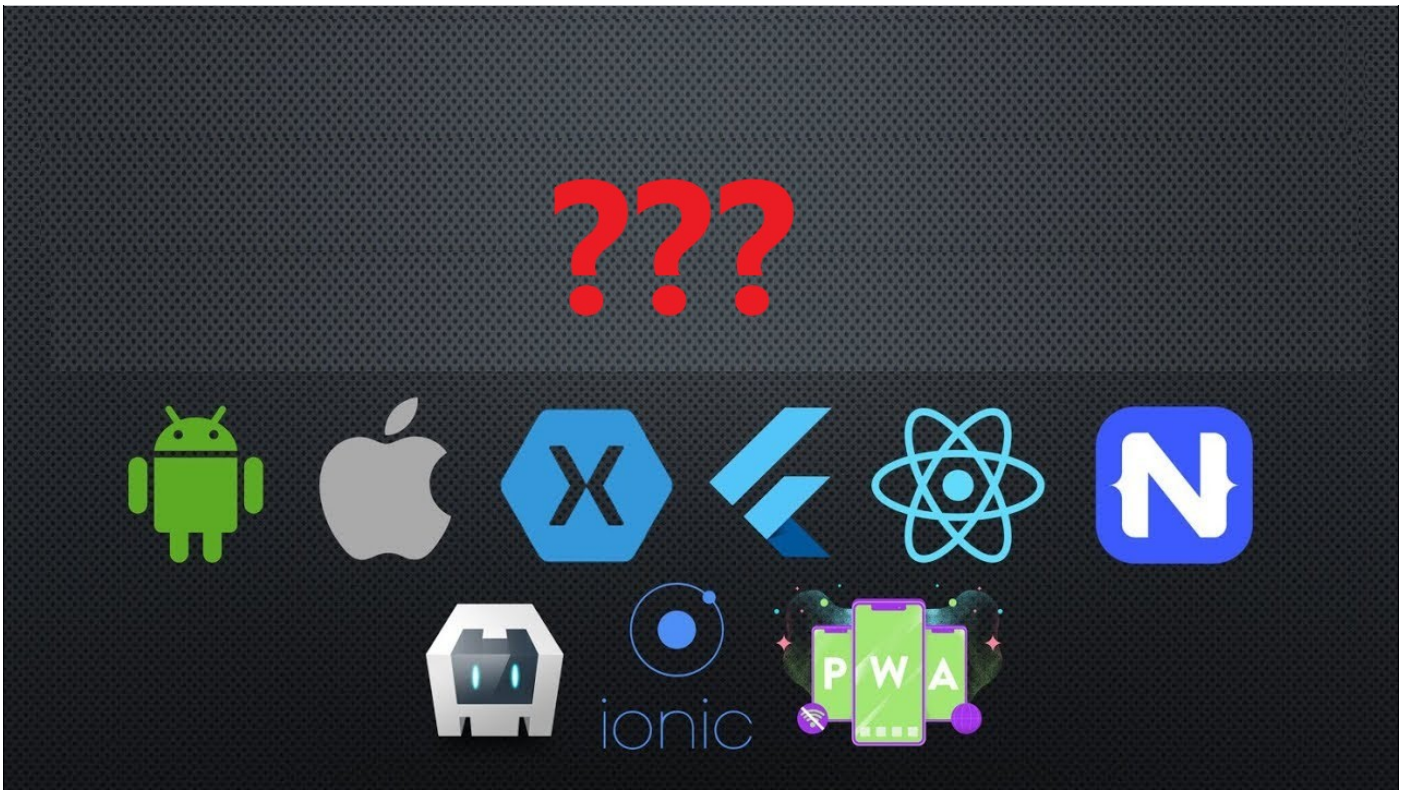
Для создания расположения или иерархии видов в вашем приложений, вы, как правило, используете *контроллер видов*. Управление видами согласовывает отображение видов, обеспечивает функциональными возможностями во взаимодействии пользователя и может обеспечивать переходы от одного экрана к другому. Например, Настройки используют контроллер навигации для отображения соподчинённости видов.

Здесь приводится пример того, как виды и контроллеры видов могут объединяться для представления пользовательского интерфейса в приложении iOS.



[iOSManual]

Виды мобильных приложений



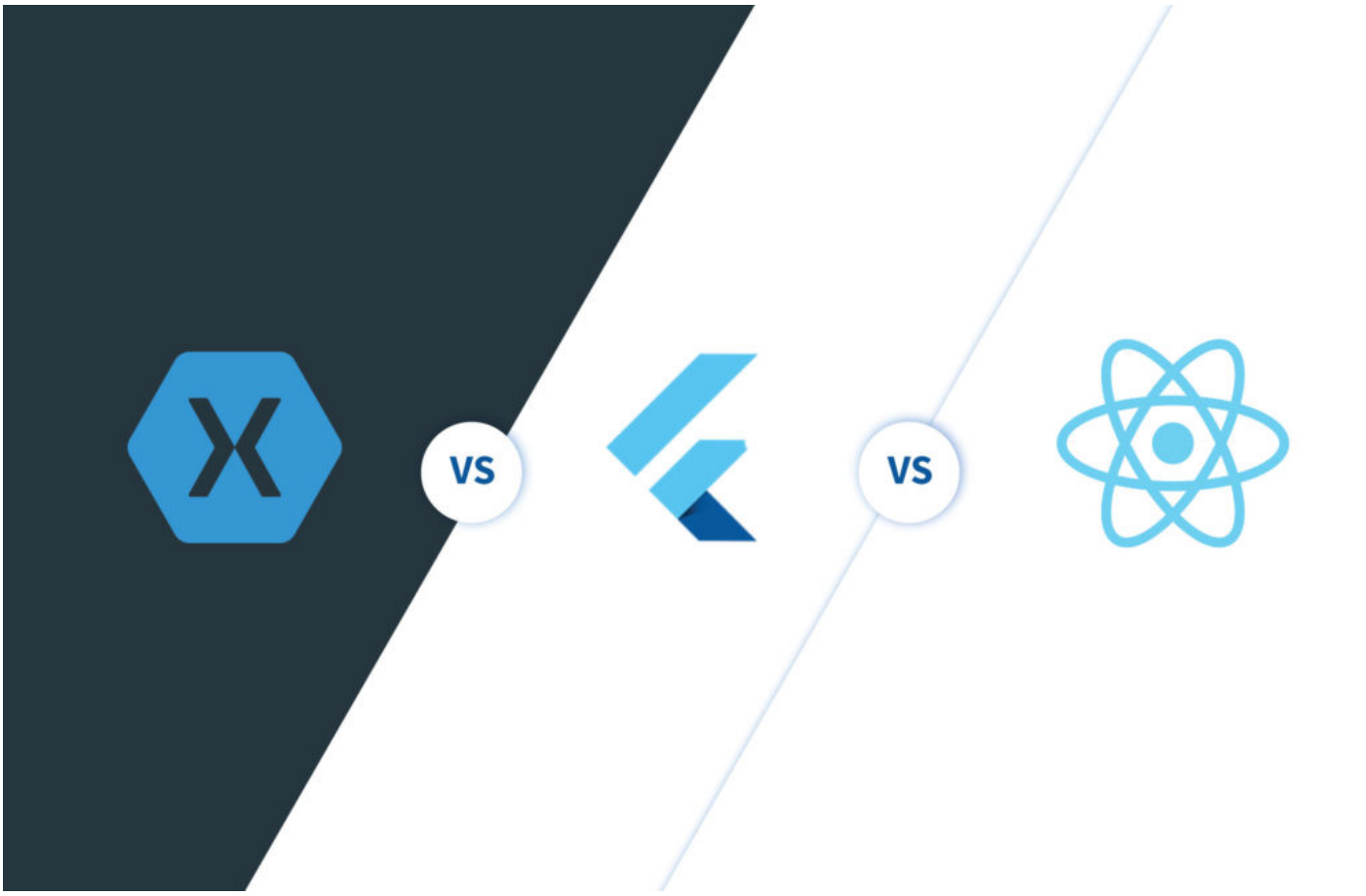
Существует три подхода технической реализации приложений для мобильных устройств:

Мобильное native-приложение — это специально разработанное приложение под конкретную мобильную платформу (iOS, Android). Такое приложение разрабатывается на языке высокого уровня и компилируется в native-код ОС, обеспечивающий максимальную производительность. Главным недостатком мобильных приложений этого типа является низкая переносимость между мобильными платформами.

Мобильное web-приложение — специализированный web-сайт, адаптированный для просмотра и функционирования на мобильном устройстве. Такое приложение хоть и не зависит от платформы, однако требует постоянного подключения к сети, т. к. физически размещено не на мобильном устройстве, а на отдельном сервере.

Гибридное приложение — мобильное приложение, "упакованное" в native-оболочку. Такое приложение, как и native, устанавливается из онлайн-магазина и имеет доступ к тем же возможностям мобильного устройства, но разрабатывается с помощью сторонних средств, использующих web-подход и такие языки как JavaScript\HTML5 и соответствующие библиотеки. В отличие от native-приложения является легко переносимым между различными платформами, однако несколько уступает в производительности и как иногда бывает с некоторым ограниченным функционалом …

3.1 Кроссплатформенное мобильное ПО



Необходимо знать почему приложение может называется кроссплатформенным. Дело в том, что, на самом деле, сам исполняющий файл, скомпилированный под одну из мобильных платформ, не может быть запущен в другой ОС. То есть, если было разработано кроссплатформенное решение и скомпилировано под Android, это не означает, что мы можем взять файл с расширением «*.apk» и запустить его на iPad'е. Идея кроссплатформенных решений не в удобстве для пользователя, а в оптимизации процессов разработки мобильного приложения. Следовательно, кроссплатформенное приложение - это решение разработанное таким образом, чтобы иметь возможность, с минимальными усилиями, скомпилировать исходный код для исполнения на разных мобильных платформах, но результатом каждой отдельной компиляции будут отдельные исполняемые файлы. Например, под iOS исполняемый файл имеет расширение - «*.ipa», под Android - «*.apk» и прочее.

--- Я считаю (на середину 2019-го) что есть только три платформы достойные внимания и это Flutter, Xamarin, ReactNative ---

3.2 Xamarin

- У него большая история, когда-то он был платным потом бесплатным, но мы начнем с той поры когда его купил Майкрософт и Xamarin умер как и Nokia (шутка), никто не умер (кроме Nokia) но и назвать эту платформу трендовой или очень популярной тоже нельзя …

Ещё в начале 2016 года разработчики тратили на лицензию Xamarin до $999 в год. Но после поглощения компанией Microsoft большая часть продуктов Xamarin стала доступна в любых версиях Visual Studio.

Стоит учитывать стоимость оборудования. Вам понадобятся устройства с macOS для разработки на iOS SDK.

- Вообщем это инструмент для кроссплатформенной разработки мобильных приложений который предлагает несколько подходов (инструментов):

Xamarin.Forms - это инструмент для разработки единого интерфейса для всех платформ (Android; iOS; Windows 10). Дизайн описывается в XML-файле, используя синтаксис XAML.

Проблема в том, что программисту будет доступен всего лишь небольшой набор стандартных элементов управления, внешняя оболочка, связанная с нативными кнопками, чекбоксами и прочим. Теоретически, в Xamarin.Forms используется до 100% общего кода, но только для приложений с совсем уж простым интерфейсом, вроде «Hello world». Если вам захочется что-то, чего нет в наборе, то придётся перерабатывать и сами элементы на каждой платформе по-отдельности.

Выбирайте Xamarin.Forms для проектов, где вы готовы сэкономить на гибкости настройки интерфейса.

Xamarin.iOS и Xamarin.Android. Оба они созданы на базе *Mono*, версии .NET Framework с открытым исходным кодом. Mono работает практически на всех платформах, включая Linux, Unix, FreeBSD и Mac OS X.

На платформе iOS компилятор Xamarin *Ahead-of-Time* (*AOT*) компилирует приложения Xamarin.iOS непосредственно в машинный код сборки ARM. На платформе Android компилятор Xamarin компилирует в *промежуточный язык* (*IL*), который при запуске приложения претерпевает *Just-in-Time*-компиляцию (*JIT*) в машинную сборку.

В обоих случаях приложения Xamarin используют среду выполнения, которая автоматически обрабатывает такие процессы, как выделение памяти, сборка мусора, взаимодействие с базовой платформой и прочее. Результатом компиляции приложений Xamarin является пакет приложения — IPA-файл в iOS или APK-файл в Android. Эти файлы неотличимы от пакетов приложений, созданных с помощью нативных сред разработки, и развертываются совершенно одинаково.
При создании приложений мы можем использовать платформу .NET и язык программирования C# (а также F#), который является достаточно производительным, и в тоже время ясным и простым для освоения и применения …

[MICROSOFT-XAMARIN]

Преимущества Xamarin:

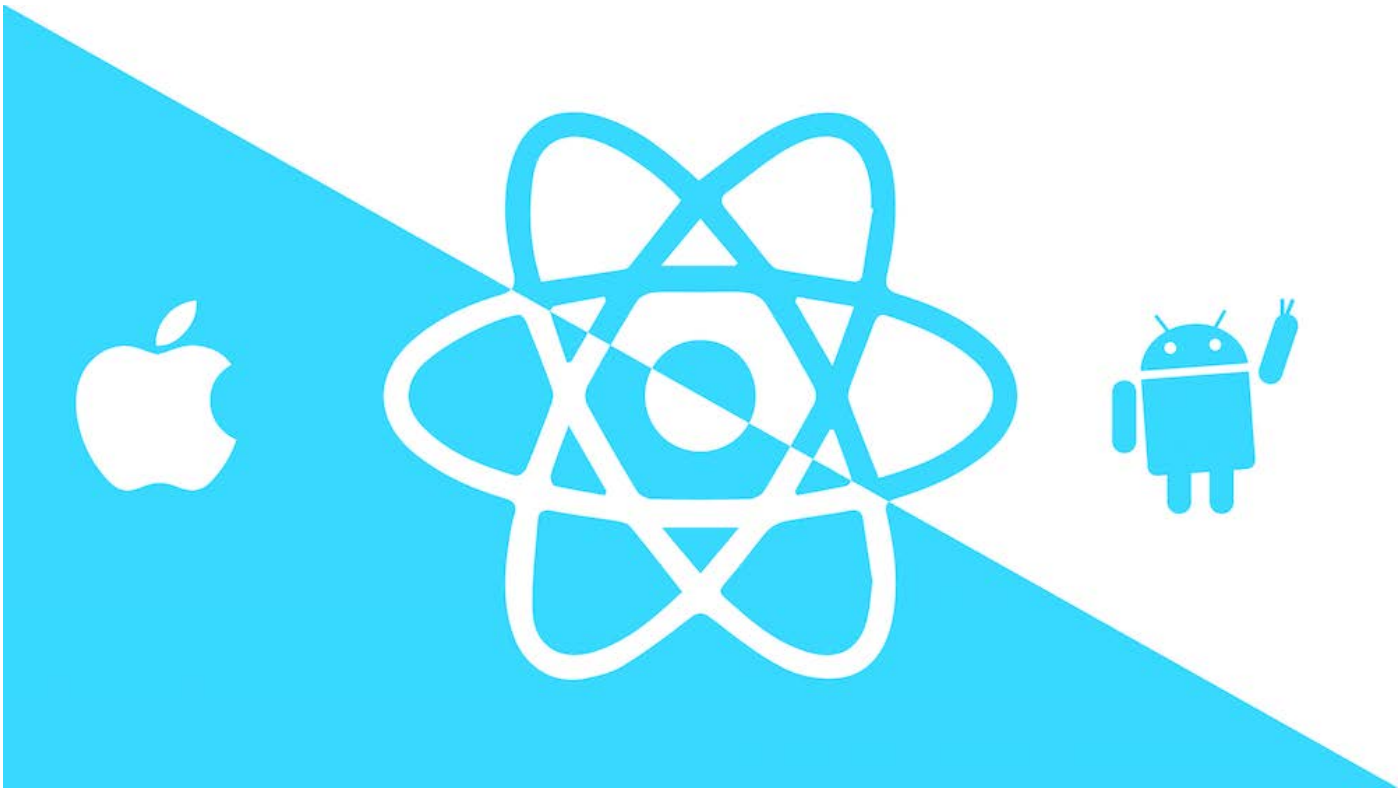
1) **Единый стек технологий для разработки на всех платформах (C# .NET)**
2) **Высокая производительность**
3) **Хорошая документация**

Недостатки Xamarin:

1) Обновление платформы довольно медлительное
2) Большой вес приложения
3) Ошибки Mono и различия с платформой .NET
4) Доступны не все существующие библиотеки
5) Высокий порог входа с нуля или разработчиков не из экосистемы .NET

--- Использование платформы Xamarin является оптимальным, если разрабатываемое приложение не является вычислительным и ресурсоемким, в сумме займет меньше времени, по сравнению с разработкой нативного приложения под каждую платформу (iOS, Android), Если же приложение сложное, ресурсоемкое, то лучше разработать нативное приложение под каждую платформу ---

3.2 ReactNative



ReactNative (RN) - это фреймворк для разработки кроссплатформенных приложений. Он даёт возможность создавать и использовать компоненты точно так же, как обычно это делается в React, вот только рендериться они будут не в HTML, а в нативные контролы операционной системы, под которую будет собрано наше приложение.

- 1) появился в начале 2015 года (детище Facebook)
- 2) построен на базе React
- 3) не использует WebView и HTML-технологии
- 4) нативные компоненты имеют биндинги в JS и обернуты в React
- 5) поддержка iOS лучше, чем Android.

- нет HTML, используется JSX
- нет CSS, используется CSS-like полифилы
- нет DOM API.
- ES6/ES7/ES8 и всё, что может babel, но нет JIT (на iOS)

Многие разработчики сталкиваются с проблемой установки и настройки существующих зависимостей ReactNative, особенно для Android. С помощью Create ReactNative App нет необходимости использовать XCode или Android Studio, и вы можете разрабатывать для своего iOS-устройства, используя Linux или Windows. Это достигается при помощи приложения **Expo**, которое загружает и запускает проекты CRNA, написанные на чистом JavaScript без компиляции любого собственного кода.

JSC (JavaScriptCore) – это JavaScript-движок, основанный на WebKit. Его использует React Native для преобразования JavaScript в машинный код. Поэтому все, что вы пишете на JavaScript, по-прежнему выполняется как JavaScript, только при помощи JSC.

JSX – это препроцессор, который добавляет синтаксис XML к JavaScript. JSX – синтаксис, похожий на XML / HTML, используемый в React, расширяет ECMAScript, так что XML / HTML-подобный текст может сосуществовать с кодом JavaScript / React. Синтаксис предназначен для использования препроцессорами (т. е. транспилерами, такими как Babel), чтобы преобразовать HTML-подобный текст, найденный в файлах JavaScript, в стандартные объекты JavaScript, которые будут анализировать движок JavaScript.

В основном, используя JSX, вы можете писать сжатые структуры HTML / XML (например, DOM подобные древовидные структуры) в том же файле, что и код JavaScript, а затем Babel преобразует эти выражения в код JavaScript. В отличие от прошлого, вместо того, чтобы помещать JavaScript в HTML, JSX позволяет нам помещать HTML в JavaScript.

Expo - очень упрощает процесс разработки мобильных приложений. Этот набор инструментов позволяет вам строить мобильное приложение не зависимо от платформы (операционной системы). Еще одним плюсом является то, что Expo предоставляет многие нативные API из коробки (например доступ к камере, иконкам, и многое другое). Таким образом, вам не придётся беспокоиться о дополнительных установках. Всё просто будет работать.

React Native Elements - это библиотека интерфейсов, которая позволит вам создавать красивые приложения просто и быстро.

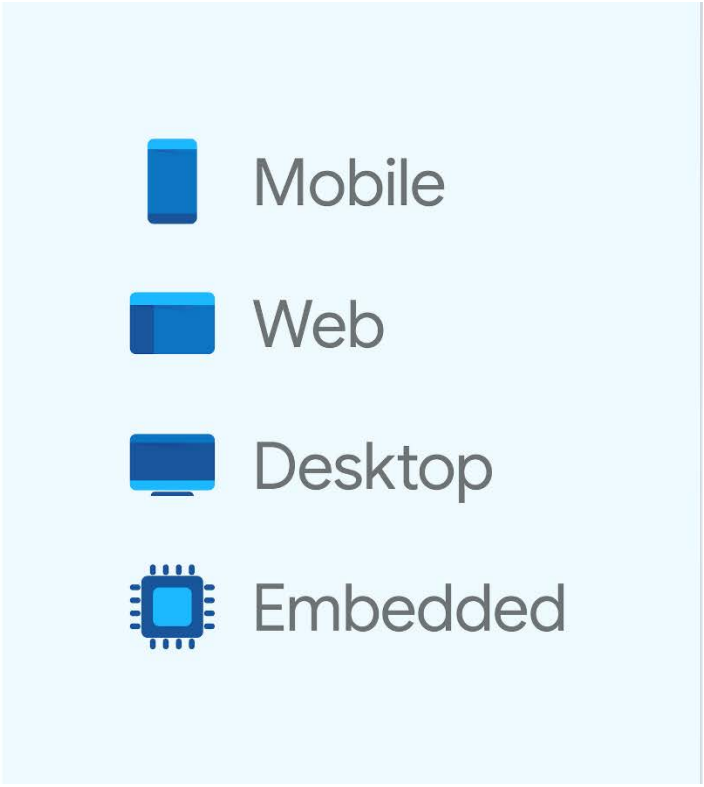
Преимущества ReactNative:

1) **Хороший выбор для тех, кто работал с Front-End'ом**
2) **Довольно быстро можно создать прототип**
3) **Хорошая документация**

Недостатки ReactNative:

1) Набит «глюками» под завязку !
2) Неадекватно работает под Android
3) JavaScript – не каждому по вкусу в мобильной разработке ...
4) Очень тяжело получить на выходе приложение которое бы работало как «нативное» по производительности !

Flutter – best for mobile developers !



- Сейчас Flutter предлагает возможность разрабатывать не только для мобильных устройств, но и для WEB, Desktop и Embedded. Так же это единственная платформа для разработки приложений для операционной системы Google Fuchsia, которая скорее всего будет использоваться в умных устройствах.

Flutter – это будущее разработки ПО …

Объективно :

Flutter — SDK с открытым исходным кодом для создания мобильных приложений от компании Google. Стабильный релиз версии 1.0 был представлен 4-го декабря 2018 года, разработка велась с 2015-го. Он используется для разработки приложений под Android и iOS, а также это пока единственный способ разработки приложений под Google Fuchsia.

Архитектура:

- Платформа Dart
- Движок Flutter
- Библиотека Foundation
- Набор виджетов

Приложения на Flutter пишутся на объектно-ориентированном языке Dart (так же создан компанией Google в 2011 году как «убийца» JavaScript).

Движок написан преимущественно на C++, он поддерживает низкоуровневый рендеринг с помощью графической библиотеки Google Skia который используется в Android. А также имеет возможность взаимодействовать с платформозависимыми SDK под Android и iOS.

Библиотека Foundation - написанная на языке Dart, содержит основные классы и методы для создания приложений Flutter и взаимодействия с движком Flutter.

Центральным элементом приложения на Flutter являются **виджеты***. Фактически это те визуальные компоненты, из которых состоит графический интерфейс.*

- В отличии от многих известных на сегодняшний день мобильных платформ, Flutter **не использует JavaScript** ни в каком виде. В качестве языка программирования для Flutter выбрали Dart, который компилируется в бинарный код, за счет чего достигается скорость выполнения операций сравнимая с Objective-C, Swift, Java, или Kotlin.
- Flutter **не использует нативные компоненты**, опять же, ни в каком виде, так что не приходится писать никаких прослоек для коммуникации с ними. Вместо этого, подобно игровым движкам, он рендерит весь интерфейс самостоятельно. Кнопки, текст, медиа-элементы, фон — все это рендерится внутри графического движка в самом Flutter. После вышесказанного стоит отметить, что “Hello World” приложение на Flutter занимает совсем немного места: iOS ≈ 2.5Mb и Android ≈ 4Mb (Xamarin около 16-ти Mb).
- Для построения UI во Flutter **используется декларативный подход**, вдохновленный веб-фреймворком ReactJS, на основе виджетов (в мире веба именуемых компонентами). Для еще большего прироста в скорости работы интерфейса **виджеты перерисовываются по необходимости** — только когда в них что-то изменилось (Stateful виджеты).
- В дополнение ко всему, в фреймворк **встроен Hot-reload**, далее на практике мы увидим как это работает.

Flutter v1.2

- **Инспектор виджетов**. Он визуализирует и помогает исследовать древовидную структуру, которую Flutter использует для рендера интерфейса.
- **Отображение шкалы времени**. Помогает исследовать приложение «кадр за кадром», выявлять те этапы рендера и вычислений, которые захламляют анимацию приложения.
- **Отладка на уровне исходного кода**. Позволяет проверять код напрямую, добавлять точки останова и изучать стек вызовов.
- **Отображение логов**. Показывает всю активность в приложении.

Flutter v1.5

Выпуск **Flutter** 1.5 поставляется с рядом обновлений для iOS и виджетов, расширенной поддержкой работы в сети и offline режиме, рядом важных обновлений для плагинов и инструментов, а также исправлениями ошибок во Flutter 1.2. тем не менее, наиболее востребованной новой функцией в флаттере 1.5 является плагин для покупки в приложениях, который теперь доступен в бета-версии для Android и IOS.

Flutter v1.7

1) Поддержка AndroidX для новых приложений
2) Поддержка Android app bundles и 64-битных приложений под Android

Начиная с 1 августа 2019 года приложения под Android, которые используют нативный код и используют Android 9 Pie должны будут предоставлять 64-битную версию в дополнение к 32-битной. Flutter уже поддерживает создание 64-битных Android приложений. Но, начиная с версии 1.7 добавлена поддержка для создания Android App Bundles сразу для 2 версий.

3) Новые виджеты

В этом релизе добавлен новый слайдер для диапазонов (скорее всего для использования в их температурных дивайсах).

4) Шрифты

В новом релизе улучшена работа с текстом для iOS. Сделан большой апгрейд поддержки rich typography — включая разные стили цифр, наборы стилей и т.п. Пример на github.

5) Увеличено количество примеров из которых можно создать код

В отличие от *PhoneGap, Cordova, Ionic* и других веб-фреймворков, Flutter не рендерит пользовательский интерфейс на WebView. **За визуализацию приложения отвечает собственный графический движок** и несколько пакетов кастомизируемых виджетов. Благодаря этому, стало возможно реализовать практически любые фантазии дизайнера интерфейсов в обозримые сроки.

В отличие от *React Native*, **языком разработки Flutter-приложений является Dart**, а не JavaScript. Dart компилируется в нативный код той платформы, для которой готовится сборка. В этом и есть залог производительного взаимодействия с платформой. Когда дело касается динамических UI-элементов, анимаций, переходов между экранами - всё должно работать предельно эффективно. К сожалению, React Native не всегда может обеспечить частоту обновления интерфейса в 60 кадров в секунду (именно при такой частоте смены кадров человеческий мозг воспринимает картинку идеально плавной). Для Flutter это не является проблемой.
В отличие от *Kotlin Native*, который сам по себе весьма любопытен, но позволяет унифицировать между платформами лишь код бизнес-логики, **Flutter позволяет единожды написать не только бизнес-логику, но и UI**. Это особенно важно, учитывая, что у типичного фронт-энд приложения, которым любое мобильное приложение по сути и является, наиболее сложна в реализации именно часть, касающаяся пользовательского интерфейса. Вместе с тем, **возможности Flutter соответствуют возможностям поддерживаемых платформ**. Благодаря технологии платформенных каналов становится возможно обращение из Dart-кода напрямую к открытым интерфейсам платформы, что может пригодиться при работе с камерой, GPS, сенсорами, файловой системой устройства и т.д.

Flutter-индустрия сейчас развивается так интенсивно, как никогда раньше, что сказывается и на ситуации на рынке труда. Низкий порог вхождения, обусловленный относительной простотой фреймворка и привычностью языка (Dart является объектно-ориентированным и по синтаксису напоминает смесь Java и JS), привлекает всё больше специалистов из смежных сфер (Android, iOS, web).

[SURF]

ЧАТ **SLACK**

E-Mail: dronwork.info@mail.ru