Dart

Основные концепты языка

Все, что вы можете поместить в переменную, является объектом, а каждый объект - экземпляром класса. Четные числа, функции и null являются объектами. Все объекты наследуются от класса Object.

Dart строго типизирован, аннотации типов необязательны, поскольку Dart может определять типы. Если вы хотите явно сказать, что никакого типа не ожидается, используйте специальный тип dynamic.

Dart поддерживает универсальные типы, например List <int> (список целых чисел) или List <dynamic> (список объектов любого типа).

Dart поддерживает функции верхнего уровня (такие как main ()), а также функции, привязанные к классу или объекту (статические методы и методы экземпляра соответственно). Вы также можете создавать функции внутри функций (вложенные или локальные функции).

Точно так же Dart поддерживает переменные верхнего уровня, а также переменные, привязанные к классу или объекту (статические переменные и переменные экземпляра). Переменные экземпляра иногда называют полями или свойствами.

В отличие от Java, в Dart нет ключевых слов public, protected и private. Если идентификатор начинается с подчеркивания (_), он является частным для своей библиотеки.

Идентификаторы могут начинаться с буквы или символа подчеркивания (_), за которым следует любая комбинация этих символов и цифр.

В Dart есть как выражения (которые имеют значения времени выполнения), так и операторы (которые не имеют). Например, условное выражение condition? expr1: expr2 имеет значение expr1 или expr2. Сравните это с выражением if-else, которое не имеет значения. Утверждение часто содержит одно или несколько выражений, но выражение не может напрямую содержать утверждение.

Инструменты Dart могут сообщать о двух типах проблем: предупреждения и ошибки. Предупреждения - это просто признаки того, что ваш код может не работать, но они не препятствуют выполнению вашей программы. Ошибки могут быть либо во время компиляции, либо во время выполнения. Ошибка времени компиляции вообще препятствует выполнению кода; ошибка времени выполнения приводит к возникновению исключения во время выполнения кода.

```
var name = 'Bob’;
dynamic name = 'Bob’;
String name = 'Bob';
```

Переменные

Неинициализированные переменные имеют начальное значение null.

Даже переменные с числовыми типами изначально равны нулю, потому что числа, как и все остальное в Dart, являются объектами.

```
int lineCount;
```

Final и const

Если вы никогда не собираетесь изменять переменную, используйте final или const либо вместо var, либо в дополнение к типу. Последняя переменная может быть установлена только один раз; переменная const - это константа времени компиляции. (Постоянные переменные неявно являются final.) Последняя переменная верхнего уровня или класса инициализируется при первом использовании.

```
final name = 'Bob'; // Without a type annotation
final String nickname = 'Bobby';
```

Встроенные типы

Тип данных определяет, какие именно значения может хранить объект. Dart имеет ряд примитивных встроенных типов:

логический тип bool

Числовые типы (int, double)

Строковые типы (String, Runes)

Symbol

bool

Тип **bool** представляет два значения: **true** и **false**.

```
var isEnabled = false;
bool isAlive = true;
```

Int

Тип **int** представляет целые числа, которые занимают не более 64 бит, точный размер зависит от платформы. Например, для Dart VM значения варьируются от -2^{63} до $2^{63} - 1$. Dart, который компилируется в JavaScript, использует систему числовых типов JavaScript с диапазоном значений от -2^{53} до $2^{53} - 1$.

Любые целые числа трактуются как значения типа int:

```
int x = 8;

var y = -5;
```

Double

Тип **double** представляет числа с плавающей точкой, которые занимают в памяти 64 бита.

Все дробные числа с точкой в качестве разделителя целой и дробной части трактуются как значения типа double:

```
double x = 8.8;

var y = -5.3;

var z = 0.09;
```

Переменной типа double также можно присвоить целочисленное значение, в этом случае оно автоматически преобразуется в дробное:

```
double x = 8;   // x = 8.0
print(x);   // 8.0
```

String

Строки представлены типом **String**, который представляет последовательность символов в кодировке UTF-16. Для определения строки можно использовать одинарные и двойные кавычки:

```
String tom = "Tom";
String sam = 'Sam';
var kate = "Kate";
var alice = 'Alice’;
```

Если необходимо определить многострочную строку, то она заключается в тройные кавычки:

```
 var multiline = '''
        Многострочная
        строка
    ''';
```

С помощью интерполяции мы можем вводить в строку значения других переменных:

```
String name = "Tom";

int age = 35;

String info = "Name: $name  Age: $age";
```

Для ввода значений в строку перед переменной ставится знак доллара $ ($name, $age), в итоге вместо $name будет вставляться значение переменной name, а вместо $age - значение переменной $age.

Runes

Тип **Runes** также представляет строки, но в отличие от String, Runes - это последовательность символов в кодировке UTF-16. Поскольку по умолчанию все строки в кавычках (как одинарных, так и двойных) представляют тип String, то для определения переменной Runes требуется специальный синтаксис:

```
Runes input = new Runes('\u041F\u0440\u0438\u0432\u0435\u0442');
    Runes text = new Runes('Привет');
    print(text);
    print(String.fromCharCodes(input));
    print(String.fromCharCodes(text));
```

Для определения значения типа Runes применяется конструктор данного типа. То есть в начале идет оператор **new**, затем название типа - Runes, после которого в скобках указывается передаваемый конструктору аргумент. Мы можем передать в конструктор либо набор кодовых единиц в кодировке UTF-16 (new Runes('\u041F\u0440\u0438\u0432\u0435\u0442')), либо обычный набор символов (new Runes('Привет')).

Чтобы получить текстовое представление объекта Runes, применяется метод **String.fromCharCodes**.

Symbol

Тип **Symbol**, наименее используемый из всех типов, представляет символьные идентификаторы, которые, как правило, применяются для ссылки на какие-то элементы API, например, библиотеки и классы. Для определения объекта этого типа применяется симол решетки #:

```
Symbol libName = #foo_lib;

var className= #foo;
```

List

List<T> или список представляет набор значений. В других языках программирования ему соответствуют массивы. Определим список чисел:

```
var list = [1, 2, 3];
```

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

Фиксированные и нефиксированные списки

Списки могут быть фиксированными (с жестко определенным размером) и нефиксированные (могут увеличиваться в размерах). Примеры создания нефиксированных списков:

```
var list1 = [];
```

```
var list2 = [2, 4, 6];
```

Свойства и методы списков

Основные свойства списков:

first: возвращает первый элемент

last: возвращает последний элемент

length: возвращает длину списка

reversed: возвращает список, в котором все элементы расположены в противоположном порядке

isEmpty: возвращает true, если список пуст

Основные методы списков:

add(E value): добавляет элемент в конец списка

addAll(Iterable<E> iterable): добавляет в конец списка другой список

clear(): удаляет все элементы из списка

indexOf(E element): возвращает первый индекс элемента

insert(int index, E element): вставляет элемент на определенную позицию

remove(Object value): удаляет объект из списка (удаляется только первое вхождение элемена в список)

removeAt(int index): удаляет элементы по индексу

removeLast(): удаляет последний элемент списка

forEach(void f(E element)): производит над элементами списка некоторое действие, которое задается функцией-параметром, аналоги цикла for..in

sort(): сортирует список

sublist(int start, [int end]): возвращает часть списка от индекса start до индекса end

contains(Object element): возвращает true, если элемент содержится в списке

join([String separator = ""]): объединяет все элементы списка в строку. Можно указать необязательный параметр separator, который будет раздлять элементы в строке

skip(int count): возвращает коллекцию, в которой отсутствуют первые count элементов

take(int count): возвращает коллекцию, которая содержит первые count элементов

where(bool test(E element)): возвращает коллекцию, элементы которой соответствуют некоторому условию, которое передается в виде функции

Set

Класс Set представляет неупорядоченный набор уникальных объектов. Для создания Set применяются фигурные скобки {}:

```
var set = {1, 2, 3, 5};

// эквивалентные определения Set

Set<int> set1 = {1, 2, 3, 5};

var set2 = <int> {1, 2, 3, 5};

Set<int> set3 = <int> {1, 2, 3, 5};
```

Отличительной особенностью Set является то, что они содержать только уникальные значения, то есть мы не можем добавить одни и те ж значения в набор несколько раз:

Свойства и методы класса Set

Основные свойства наборов:

first: возвращает первый элемент

last: возвращает последний элемент

length: возвращает длину набора

isEmpty: возвращает true, если набор пуст

Основные методы наборов:

add(E value): добавляет элемент в набор

addAll(Iterable<E> iterable): добавляет в набор другую коллекцию

clear(): удаляет все элементы из набора

difference(Set<Object> other): возвращает разность текущего набора и набора other в виде другого набора

intersection(Set<Object> other): возвращает пересечение текущего набора и набора other в виде другого набора

remove(Object value): удаляет объект из набора

removeAll(Iterable<Object> elements): удаляет из набора все элементы коллекции elements

union(Set<E&g; other): возвращает объединение двух наборов - текущего и набора other

contains(Object element): возвращает true, если элемент содержится в наборе

join([String separator = ""]): объединяет все элементы набора в строку. Можно указать необязательный параметр separator, который будет раздлять элементы в строке

skip(int count): возвращает коллекцию, в которой отсутствуют первые count элементов

take(int count): возвращает коллекцию, которая содержит первые count элементов

where(bool test(E element)): возвращает коллекцию, элементы которой соответствуют некоторому условию, которое передается в виде функции

Map

Класс Map представляет коллекцию элементов, где каждый элемент имеет ключ и значение. Ключ и значение элемента могут представлять различные типы.

Для создания объекта Map могут использоваться фигурные скобки {}, внутри которых помещаются пары ключ-значение, либо конструктор класса Map. Варианты создания Map:

```
var map = {
    1: "Tom",
    2: "Bob",
    3: "Sam"
};
// эквивалентное определение Map
Map<int, String> map2 = {
    1: "Tom",
    2: "Bob",
    3: "Sam"
};
```

Свойства и методы Map

Основные свойства:

entries: возвращает объект Iterable<MapEntry<K, V>>, который представляет все элементы Map

keys: возвращает объект Iterable<K>, который представляет все ключи Map

values: возвращает объект Iterable<V>, который представляет все значения Map

length: возвращает количество элементов в Map

isEmpty: возвращает true, если Map пуст

Основные методы Map:

addAll(Map<K, V> other): добавляет в Map другой объект Map

addEntries(Iterable<MapEntry<K, V>> newEntries): добавляет в Map коллекцию Iterable<MapEntry<K, V>>

clear(): удаляет все элементы из Map

containsKey(Object key): возвращает true, если Map содержит ключ key

containsValue(Object value): возвращает true, если Map содержит значение value

remove(Object key): удаляет из Map элемент с ключом key

Функции

```
[тип] имя_функции(параметры){

    выполняемые_выражения
}
```

Если функция состоит из одного выражения, то ее можно сократить следующим образом:

```
[тип] имя_функции(параметры) => выполняемое_выражение
```

Необязательные параметры функции

Ряд параметров мы можем сделать необязательными, то есть мы можем не передавать для них никаких значений. Для этого параметр заключается в квадратные скобки. Но при этом ему необходимо предоставить значение по умолчанию, которое будет использоваться, если параметру не передано никакого значения:

```
void showPerson(String name, [int age = 22]){
    print("Name: $name");
    print("Age: $age \n");
}
```

Именованные параметры

Dart поддерживает передачу параметров по имени:

```
void main() {

    showPerson(name: "Tom", age: 35);
    showPerson(age: 29, name: "Alice");
    showPerson(name: "Kate");
}
void showPerson({String name = "undefined", int age=0}){
    print("Name: $name");
    print("Age: $age \n");
}
```

Константы в функциях

Параметры позволяют передать значения локальным переменным, которые определены в функциях. Но также можно передавать через параметры значения локальным **final**-константам, поскольку они определяются во время выполнения. Но при этом мы не можем передавать значения **const**-константам, так как их значения должны известны уже во время компиляции. В связи с чем при разных вызовах функции мы можем передать **final**-константам разные значения

```
void main() {
showPerson("Alice", 22);
}
void showPerson(String userName, int userAge){
    // const int initialAge = userAge; - так нельзя, константа должна определяться на этапе компиляции
    final String name = userName;
    final int age = userAge;
    print("Name: $name   Age: $age \n");
}
```

Функция как объект

Любая функция в языке Dart представляет тип **Function** и фактически может выступать в качестве отдельного объекта. Например, мы можем определить объект функции, присвоить ему динамически ссылку на какую-нибудь функцию и вызвать ее

```
void main() {
    Function func = hello;
    func();     // Hello!
    func = bye;
    func();     // Goodbye!
}
void hello(){
    print("Hello!");
}
void bye(){
print("Goodbye!");
}
```

Функции могут выступать в качестве параметров других функций:

```
void main() {
showMessage(hello);
   showMessage(bye);
}
void showMessage(Function func){
   func();    // вызываем переданную функцию
}
void hello(){
   print("Hello!");
}
void bye(){
print("Goodbye!");
}
```

функция может выступать в качестве возвращаемого значения

```
void main() {
Function message = getMessage(11);
    message();
    message = getMessage(15);
    message();
}

Function getMessage(int hour){
    if(hour < 12) return morning;
    else return evening;
}
void morning(){
    print("Good morning!");
}
void evening(){
    print("Good evening!");
}
```

Анонимные функции

Анонимные функции похожи на обычные функции за тем исключением, что они не имеют названия

```
Function operation = (a, b) => a + b;
```

Использование: если вызываем функцию, которая принимает другую функцию, то для функции-параметра может быть проще использовать анонимную функцию, чем определять именованную, особенно если именованная функция нам больше нигде не понадобится

Вложенные функции

В Dart функции могут быть вложенными в другие функции

```
void main() {
    void hello(){
        print("Hello!");
    }
    hello();
    hello();
}
```

Причем определение вложенной функции должно идти до ее вызова. Вложенные функции имеют смысл, если мы планируем использовать некие повторяющиеся действия только внутри определенной функции.

Причем вложенные функции сами могут содержать другие вложенные функции

Стоит отметить, что вложенные функции образуют собственный контекст - переменные и константы, к которым внешняя функция не может обратиться.

Closure

Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости.

Технически замыкание включает три компонента:

внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные и параметры - лексическое окружение

переменные и параметры (лексическое окружение), которые определены во внешней функции

вложенная функция, которая использует переменные и параметры внешней функции

```
Function outer(){       // внешняя функция
   var n;               // некоторая переменная - лексическое окружение
   Function inner(){    // вложенная функция
      // действия с переменной n
   }
   return inner;
}
```

Классы и объекты

Dart является объектно-ориентированным языком, и каждое значение, которым мы манипулируем в программе на Dart, является объектом.

Объект представляет **экземпляр** некоторого класса, а **класс** является шаблоном или описанием объекта. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова **class**:

```
class Person{


}
```

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы.

Конструкторы

Кроме обычных методов классы могут определять специальные методы, которые называются **конструкторами**. 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.

Именованные конструкторы

По умолчанию мы можем определить только один общий конструктор. Если же нам необходимо использовать в классе сразу несколько конструкторов, то в этом случае нужно применять **именованные конструкторы** (named constructors).

```
class Person{
    Person.undefined(){
    }
    Person.fromName(String n){
    }
    Person(String n, int a)
    {
}
```

Сокращенная версия констуктора

Используя ключевое слово this, мы можем сократить определение конструктор

```
class Person{

    String name;
    int age;

    Person(this.name, this.age);

    void display(){
        print("Name: $name Age: $age");
    }
}
```

Инициализаторы

Инициализаторы представляют способ инициализации полей класса

Список инициализации указывает после параметров конструктора через двоеточие до открывающей фигурной скобки

Person(userName, userAge) : name=userName, age = userAge {

Каскадная нотация

Каскадная нотация - операция .. позволяет выполнить последовательность операций над одним объектом

```
Person tom = Person()
    ..name = "Tom"
    ..age = 36
    ..display();
```

Константные конструкторы классов

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

```
class Person{

    final String name;
    final int age;

    // константный конструкор
    const Person(this.name, this.age);
}
```

Наследование

Наследование является одним из ключевых моментов объектно-ориентированного программирования, позволяя передавать одним классам функционал других. В языке Dart наследование реализуется с помощью ключевого слова **extends** (как в Java)

```
class Employee extends Person{


}
```

Конструкторы и ключевое слово super

В отличие от полей и методов конструкторы базового класса не наследуются. Если базовый класс явным образом определяет конструктор (конструктор по умолчанию не учитывается), то его необходимо вызвать в классе-наследнике при определении конструктора

```
class Employee extends Person{

    Employee(name) : super(name);
}
```

Переопределение методов

Производные классы могут определять свои поля и методы, но также могут переопределять, изменять поведение методов базового класса. Для этого применяется аннотация @override:

```
class Employee extends Person{
     String company = "";
     Employee(name, this.company) : super(name);


   @override
   void display(){
      print("Name: $name");
      print("Company: $company");
   }
}
```

Абстрактные классы и методы

Абстрактные классы представляют классы, определенные с ключевым словом **abstract**

```
abstract class Figure {

}
```

Абстрактные классы похожи на обычные классы (также могут определять поля, методы, конструкторы) за тем исключением, что мы не можем создать напрямую объект абстрактного класса, используя его конструктор.

Как правило, абстрактные классы описывают сущности, которые в реальности не имеют конкретного воплощения. Например, геометрическая фигура может представлять круг, квадрат, треугольник, но как таковой геометрической фигуры самой по себе не существует. Есть конкретные фигуры, с которыми мы и работаем. В то же время все фигуры могут иметь какой-то общий функционал, например, методы вычисления периметра, площади и т.д

Здесь абстрактный класс геометрической фигуры определяет метод calculateArea(), который выводит на консоль площадь фигуры. Класс прямоугольника определяет свою реализацию для этого метода.

Абстрактные методы

В примере выше метод calculateArea в базовом классе Figure не выполняет никакой полезной работы, так как у абстрактной фигуры не может быть площади. И в этом случае подобный метод лучше определить как абстрактный:

Абстрактный метод определяется также, как и обычный, только вместо тела метода после списка параметров идет точка с запятой: void calculateArea();.

Важно отметить, что 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.

Реализация интерфейсов

Наследование в языке Dart имеет важное ограничение: мы не можем наследовать класс сразу от нескольких классов

Для решения этой проблемы в Dart применяется реализация интерфейсов. При этом важно понимать, что интерфейс - это не отдельная сущность, как в некоторых языках программирования (например, тип interface в C# или Java), а тот же класс. То есть класс в Dart одновременно выступает в роли интерфейса, и другой класс может реализовать данный интерфейс.

Интерфейс представляет синтаксический контракт, которому должны следовать реализующие этот интерфейс классы. То есть, если класс-интерфейс определяет какие-нибудь поля и методы, то класс, реализующий данный интерфейс, должен также определить эти поля и методы.

Для реализации интерфейсов применяется оператор **implements**

```
class Person{

    String name;
    Person(this.name);

    void display(){
        print("Name: $name");
    }
}

class Employee implements Person{

    String name = "";          // реализация поля name
    // реализация метода display
    void display(){
        print("Employee name: $name");
    }
}
```

Наследование классов vs реализация интерфейсов

При наследовании производный класс не обязан определять те же поля и методы, которые есть в базовом классе (за исключением абстрактных методов). Если базовом классе определяется конструктор, то производный класс обязан определить свой конструктор, при котором вызывается конструктор базового класса. В производном классе мы можем обращаться к реализации базового класса с помощью ключевого слова **super**. Не поддерживается множественное наследование.

При реализации интерфейса производный класс должен определить все поля и методы, которые определены в классе интерфейса. Если в базовом есть конструктор, то производный класс НЕ обязан определять свой конструктор. В производном классе мы НЕ можем обращаться к методам реализованного интерфейса с помощью ключевого слова **super**. Поддерживается множественная реализация интерфейсов.

Статические члены классов

Кроме обычных методов и полей класс может иметь статические поля, методы, константы. Значения таких полей, методов и констант относятся в целом ко всему классу, а не к отдельным объектам. Для определения статических переменных, констант и методов перед их объявлением указывается ключевое слово **static**.

Статические поля

Статические поля хранят состояние всего класса. Статическое поле определяется как и обычное, только впереди ставится ключевое слово **static**

```
class Employee{
    static int retirenmentAge = 60;
}
```

Статические методы

Статические методы также относятся ко всему классу и предваряются ключевым словом static. Как правило, статические методы выполняют такие вычисления, которые не затрагивают состояние или поведение объекта:

```
class Operation{
    static int sum(int x, int y) => x + y;
    static int subtract(int x, int y) => x - y;
    static int multiply(int x, int y) => x * y;
}
```

Стоит отметить, что Google не рекомендует создавать классы, которые содержать только статические методы и переменные/константы, как выше определенный класс Operation.

В чем преимущества статических полей и методов перед нестатическими?
Статические поля и методы потребляют меньше памяти, чем нестатические. Нестатическая переменная после инициализации (при первом присвоении значения) сразу начинает потреблять память независимо от того, используется она или нет. А статические поля и методы не инициализируются до того момента, пока они не начнут использоваться в программе. Соответственно потреблять память они начинают только тогда, когда непосредственно начинают использоваться.

Generics

Generics или обобщения позволяют добавить программе гибкости и уйти от жесткой привязки к определенным типам. Иногда возникает необходимость, определить функционал таким образом, чтобы он мог использовать данные любых типов.

Generics или обобщения позволяют обеспечить большую безопасность типов и помогают избежать дублирования кода.

```
class Person<T>{
    T id;   // идентификатор пользователя
    String name; // имя пользователя
    Person(this.id, this.name);
}
```

Ограничения обобщений

С помощью выражения <T extends Account> указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению мы можем использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их полям и методам.

```
class Transaction<T extends Account>{


    T fromAccount;  // с какого счета перевод
    T toAccount;
}
```

Фабричный конструктор

По умолчанию конструкторы класса создают и возвращают новый объект этого класса. Но, возможно, не всегда потребуется создавать новый объект класса. Возможно, мы захотим использовать и возвращать из конструктора уже имеющийся объект. Для этой цели в языке Dart применяются фабричные конструкторы или конструкторы, которые предваряются ключевым словом **factory**.

```
factory Application(String name) {
        if(app.name == ""){
            app = Application.fromName(name);
            print("Приложение $name запущено");
        }
        else{
             // в фабричных конструкторах нельзя обращаться к this
            // print("В приложении ${this.name} открыта новая вкладка");
            print("В приложении ${app.name} открыта новая вкладка");
        }
        return app;
    }
```

Переопределение операторов

Dart позволяет изменить поведение ряда встроенных операторов, что позволяет нам воспользоваться дополнительными возможностями по работе с объектами.

Переопределение операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

возвращаемый_тип operator оператор(параметр) { }

Перечисления

Перечисления в Dart или enum представляют особый тип классов, который представляет фиксированный набор константных значений. Для определения перечисления применяется ключевое слово **enum**.

```
enum Operation{
    add,
    subtract,
    multiply
}
```

Блок try..catch

В процессе работы прогаммы могут возникать различные ошибки, которые нарушают привычный ход программы и даже заставляют ее прервать выполнение. Такие ошибки называются исключениями. Язык Dart, как и многие языки программирования, поддерживает обработку исключений, что позволяет нам избежать аварийного прерывания работы программы.

```
try{
    // Код, который может привести к генерации исключения
}
on Тип_Исключения{
    // Обработка возникшего исключения
}
catch (e){
    // Обработка возникшего исключения
}
finally{
    //
}
```

Разница между операторами **on** и **catch** состоит в том, что **on** обрабатывает исключение определенного типа А **catch** обрабатывает все исключения.

Ключевой класс для обработки исключений в Dart - это класс **Exception**, который является базовым классом для всех остальных типов исключений. По умолчанию стандартная библиотека предоставляет ряд встроенных типов исключений.

В конструкции try..catch мы можем обрабатывать исключения определенного типа.

После оператора on указывается тип исключения, которое надо обработать. В данном случае выражение on Exception говорит, что мы обрабатываем исключения типа Exception. А поскольку это базовый тип для всех исключений, то фактически мы обабатываем все исключения.

Оператор throw

Оператор **throw** позволяет нам сами сгенерировать исключение в необходимом месте

```
class Person{
    String name;
    int age = 1;

    Person(this.name, age){
        if(age < 1 || age > 110) {
            throw Exception("Недопустимый возраст");
        }
        else{
            this.age = age;
        }
    }
}
```