
Yandex

Yandex

Dart programming language

Sergey Koltsov, Yandex.Pro Team Lead

Table of Content

About Dart

- C-like syntax
- Object-oriented programming language
- Static type checking
- Supports JIT and AOT compiling
- Fast object allocation and garbage collection
- Single thread

Helpful links

- dart.dev
- dartpad.dev
- Language tour

Syntax

main() function

```dart
/*
    Multiline comment
*/
// Entry-point function
void main () {
  print("Hello Dart!");  // print to console
}
```

main() function

```
// Entry-point function
void main (List<String> args) {
  print("Hello ${args.first}!”);  // print to console
}
```

Variables

```
void main () {
  String text = "Hello 2021!”; // assign value to a variable
  print(text);
  text = "Bye-bye COVID-19!"; // update the value of the variable
  print(text);
}
```

Variables

```
void main () {
  var text = "Bye-bye 2021!”; // Dart infer String type automatically
  print(text);
  text = 3.14; // Exception, name is a String, not a number
  dynamic anotherText = "Bye-bye COVID-19!";
  anotherText = 3.14; // No exception, you can assign any type to a dynamic variable
  print(anotherText);
}
```

Variables

```
void main () {
  const text = "Bye-bye 2021!”;
  print(text);
  text = "Bye-bye COVID-19!"; // Exception, you cannot change const value
  final anotherText = "Bye-bye 2021!";
  print(text);
  text = "Bye-bye COVID-19!"; // Exception, you cannot change final value
  // const is ready at compile time,
  // but final will be ready only in runtime
}
```

Data types

- Numbers (int, double)
- Strings (String), Runes (Runes)
- Booleans (bool)
- Lists (List)
- Sets (Set)
- Maps (Map)

Numbers

- Integers - **int**. Uses 64 bit depending on platform
- Floating-point numbers - **double**. Uses 64 bit
- **int** and **double** are subtypes for **num.**

Num

```
void main () {
  // int declarations
  var x = 1;
  var hex = 0xDEADBEEF;
  var exponent = 8e5;
  // double declarations
  var y = 1.1;
  var exponents = 1.42e5;
  // x can be used as int or double
  num x = 1;
  x += 2.5;
}
```

Strings

```
void main () {
  // String = a sequence of UTF-16 symbols.
  var string1 = 'You can use single quote marks';
  var string2 = "You can use double quote marks";
  var string3 = '+ opreator ' + 'works for strings concatination';
  // You can use $ symbol for passing variable inside a string
  var s4 = 'string interpolation';
  var s5= 'Dart has $s4 - that is handy';
}
```

Runes

```
void main () {
  // Runes are also strings, but in Unicode.
  var s1 = 'I \u2665 Dart';
  print(s1); // Prints: 'I ♥ dart'
 }
```

Booleans

```
void main () {
  // Use bool for booleans: true and false
  var off = false;
  var on = true;
  var someIntValue = 1;
  if (someIntValue) // Exception, you can use only bool values in if-statements
 }
```

List

```
void main () {
  // List<T> - a list of objects witch has a type. Indexes start with 0.
  var list =[1, 2, 3];
  var first = list[0]; // first is 1
  var last = list[list.length -1]; // last is 3
  const constlist = [1, 2, 3]; // compile-time const value
  constlist[0] = 0; // exception, you cannot update const list
 }
```

List

```
void main () {
  var list = [-1, 3, 0, 1, 2, -2];
  print(list.first);  // prints -1
  print(list.length); // prints 6
  list.sort(); // sorts this exact list
  print(list.join(", ")); // prints -2, -1, 0, 1, 2, 3
  list.removeAt(1); // remove second element
  list = list.where((element) ⟹ element > 1).toList(); // get all items greater than 1
 }
```

Set

```dart
void main () {
  // Set<T> - an unordered collection of unique values
  var set = {1, 2, 3, 5}; // declaration
  for (var n in set) print(n);// loop for set
  set = <int>{}; // specify type explicitly, otherwise Map will be created
  set.add(1);
  print(set); // prints {1}
  set.addAll([1,2,3]); // adds a List to the Set
  print(set); // prints {1, 2, 3}
  print(set.contains(1)); // true
  print(set.intersection({1,2})); // {1, 2}
 }
```

Map

```
void main () {
  // A collection of key/value pairs, from which you retrieve a value
  // using its associated key.
  var map = {1: "first", 2: "second", 3: "third"};
  map[4] = "fourth";  // add an element
  map.containsKey(1); // the key exists, true
  map.remove(2); // remove element with the key 2
  print(map); // {1: first, 3: third, 4: fourth}
}
```

Null safety

- Support from Dart 2.12 and Flutter 2.
- Use **?** after any type for showing if the value can be **null**
- NPE are checked at compile time and not in runtime

Null safety

```
void main () {
  String name = "Sergey";
  String? surname = "Ivanov";
  name = null; // error, cannot be null
  surname = null; // ok
  var finalSurname = surname ?? "undefined";
  print(finalSurname); // "undefined"
  int? a = 10;
  int b = a!; // force-unwrap, a =/= null
  print(b) // prints 10
 }
```

Null safety

```
class Coffee {
   String? _temperature;
   void heat() { _temperature = 'hot'; }
   void chill() { _temperature = 'iced'; }
   String serve() ⇒ _temperature! + ' coffee';
  // we have to use ! every time, but value cannot be null for _temperature
}
```

Null safety

```
class Coffee {
   String _temperature = ‘normal’;
   void heat() { _temperature = 'hot'; }
   void chill() { _temperature = 'iced'; }
   String serve() ⇒ _temperature + ' coffee';
   // We guarantee that _temperature is set to non-null value
   // before it will be used for the first time
}
```

Conditions

- If … else
- switch..case
- Ternary operator ?:
- Default ??

Conditions

```
void main (int? arg) {
 int num = arg ?? 3;
 int output = 0;
 switch(num) {
    case 1:
        output = 3;
        break;
    case 2: // you can join muliple cases
    case 3:
        output = 6;
        output = output % 3 == 0 ? 2 : 5; // ternary operator
        break;
    default: // default branch is used in case other cases were false
     output = 24;
 }
}
```

Conditions

```
void main () {
  int x = 3;
  int y = 2;
  int z = x < y ? (x + y) : (x - y);
  print(z); // prints 1
 }
```

Functions

- Functions are also objects
- Support optional and named parameters
- Anonymous and nested functions

Functions

```
bool isPositive(int a) {
    return a > 0;
}

bool isPositive(int a) ⇒ a > 0; // arrow syntax, for single-line functions
```

Named parameters

```
int funWithParameters(
  int requiredArg, { // required positional parameter
  int? nullableNamedArg, // non-required named parameter
  int namedArg = 0, // non-required named parameter with default value
  required int reqNamedArg, // required named parameter
}){
  return requiredArg + namedArg;
}
```

Optional positional parameters

```
int funWithParameters(
  int requiredArg,[
  int optionalArg = 0, // optional non-named parameter
 ]){
  return requiredArg + namedArg;
}
void main (){
   funWithParameters(1, 3); // prints 4
   funWithParameters(1);   // prints 1
}
```

Function as a parameter

```
void printElement(String element) {
    print(element);
}
var list = [1, 2, 3];
list.forEach(printElement); // use printElement as a parameter
```

Anonymous functions

```
const list = ['1', '2', '3'];
list.forEach((item) { // an anonymous function that is used as a parameter for forEach
 print('${list.indexOf(item)}: $item');
});
```

Exceptions

- Try execute some code and catch an exception when execution has failed
- Key-words: throw, try, catch, on, finally

Exceptions (throw)

```
// throw an exception
throw FormatException('Expected at least 1 section');
// You can throw Error or Exception
```

Exceptions (catch)

- Wrap code with **try { }**
- Use **catch** for **any kind of Object** that has been thrown
- Use **on** for specifying type of exceptions you would like to catch
- Use **finally** for executing some code even when exception was thrown

Exceptions (catch)

```
try {
  someFun();
} on FileSystemException catch {  // no access for files
   requestPermission();
} catch (e) {                     // all other exceptions except FileSystemException
   print('Unknown exception: $e');
   rethrow;                       // throw current exception further
} finally {
   someAction();                  // always will be executed after try and catch
}
```

OOP

OOP in Dart

- God parent — Object
- No multiple inheritance
- Abstract classes
- Mixins
- Generics
- Enums

Class members

```
class Point {
   double x = 0;
   double y = 0;
   bool _privateField = false; // use underscore (_) for private fields or classes
   Point(this.x, this.y);      // short version of a constructor
   double distanceTo(Point point) {
      // implementation
   }
}
var p = Point(2, 2); // create an instance
double distance = p.distanceTo(Point(4, 4)); // calling method for a Point
var a = p?.y; // if p is nullable, use ?.
```

Constructors

```
var p1 = Point(2, 2);  // classic
var p2 = Point.fromJson({'x': 1, 'y': 2}); // named
var p3 = const ImmutablePoint(2, 2); // const: p3 is a compile-time constant
```

Inheritance

```
class CarPoint extends Point {                    // extends for using inheritance
  double direction;                               // CarPoint field
  CarPoint(this.direction, x, y) : super(x, y); // call the constructor of the parent

  @override
  double distanceTo(Point point) { // override an implementation
     //implementation
  }
}
```

Abstract classes

```
abstract class Figure {
   void calculateArea();                // an abstract method without a body
}
Figure ellipse = Figure();              // error, you cannot instantiate
                                        // an instance of an abstract class
class Rectangle extends Figure {
   int width;
   int height;
   Rectangle(this.width, this.height);
   @override
   void calculateArea() {               // child must override parent's methods
      int area = width * height;
      print("area = $area");
   }
```

Interfaces

- Any class is implicitly an interface
- You can implement any number of interfaces in you class
- Use **implements**
- An implementation must override all fields and methods

Interfaces

```
class Square implements Rectangle {
   @override                              // must override public fields
   int width;
   @override
   int height;
   Square(this.width, this.height);
   @override
   void calculateArea() {                  // must override all methods of an interface
      //implementation
  }
}
```

Mixins

- Like a class, but without constructor
- For implementation use a key-word **with**
- You can implement any number of mixins

Mixins

```
mixin Worker {
   String company = "";
   void work() {
      print("Work in $company");
   }
}
class Person with Worker {
   Person(comp) {
      company = comp;
   }
}
Person p = Person("Yandex"); p.work();
```

Static variables and methods

```
class MyClass {
   static const pi = 3.14;    // you can access pi from the name of the class
   static bool isPositive(int a) ⇒ a > 0; // no need to create an instance of MyClass
}

final radius = 5;
final length = 2 * MyClass.pi * radius;
print(MyClass.isPositive(length));
```

Enums

```
enum Color {
  red,
  green,
  blue
}

var textColor = Color.red;
```

Generics

- Helps to avoid code duplication
- Helps to keep type safety

Generics

```
1. var numbers = <int>[];
names.addAll([1, 2, 3]);
names.add(2.5); //exception

2. abstract class ObjectCache {
   Object getByKey(String key);
   void setByKey(String key, Object value);
}
abstract class StringCache {   // code duplication for different types
   String getByKey(String key);
   void setByKey(String key, String value);
}
```

Generics

```
abstract class Cache<T> {                  // single class for any type
    T getByKey(String key);
    void setByKey(String key, T value);
}

class Foo<T extends Object> {             // made T stricter, now it must be non-nullable
}
```

Generics

```
// You can use T in functions
T first<T>(List<T> ts) {
  T tmp = ts[0];
  return tmp;
}
```

Asynchronous programming

Asynchronous programming

- Single thread
- Asynchronous via Event Loop
- Two queues of events: Event Loop and microtasks

Asynchronous programming

- Future API
- Async and await
- Stream

Future

- An instance of Future<T> keeps the result of the future operation
- Two states: Uncompleted and Completed
- Completed state can have one of results: a value or an error

Future

```
Future<void> getMessage() {
   // use delayed for emulation of a long operation
   return Future.delayed(Duration(seconds: 3), () ⇒ print("2 new messages"));
}
void main() {
    getMessage();
    print("Check messages...");
}
// Result in console:
// Check messages...
// 2 new messages
```

Future

- Future.value([FutureOr<T> value])
- Future.error(Object error, [StackTrace stackTrace])
- Future.delayed(Duration duration, [FutureOr<T> computation()])
- Others

Future API

```
// HttpRequest.getString(url) - returns Future<String>
void getData(String url) {
   HttpRequest.getString(url).then((String result) {
       print(result);  // handle the result
   }).catchError((e) {
      // handle an error
   });

  // not here
}
```

async and await

```
// HttpRequest.getString(url) - returns Future<String>
Future<void> getData(String url) async { // if await keyword inside – use async
   try {
      // execution will freeze inside getData until result will be returned
      String result = await HttpRequest.getString(url)
      print(result); // handle the result
   } catch (e) {
       // handle an error
   }
}
```

Live example

```dart
import 'dart:async';

void main() {
  var counter = 0;
  final timer = Timer.periodic(Duration(seconds: 1), (_) ⇒ print(counter++));
  Future.delayed(Duration(seconds: 10)).then((_) {
    print('Finished');
    timer.cancel();
  });
}

void main() async {
  var counter = 0;
  final timer = Timer.periodic(Duration(seconds: 1), (_) ⇒ print(counter++));
  await Future.delayed(Duration(seconds: 10));
  print('Finished');
  timer.cancel();
}
```

Streams

- Sequence of async events
- Two types: single subscription and broadcast
- Can subscribe an unsubscribe from stream
- Can transform and process a stream
- Can be handled via **await for** or **listen()**
- Exceptions can be handled

await for

```
Future<int> sumStream(Stream<int> stream) async {
   var sum = 0;
   try {
      await for (final value in stream) {
         sum += value;
      }
   } catch (e) {
      return -1;   // return -1 in case of an error
   }
   return sum;
}
```

Stream types

- Single subscription — subscribe only once. For example, server request.
- Broadcast — many subscribers. For example, UI events or location.

Handle streams

- Methods of a Stream that return Future<T> — processing
- Methods of a Stream that return Stream<S> — transforming

Stream processing

- Future<T> elementAt(int index);
- Future<bool> every(bool Function(T element) test);
- Future<bool> any(bool Function(T element) test);
- Future<T> firstWhere(bool Function(T element) test, {T Function()? orElse});
- Future<T> get first;
- Future<List<T>> toList();

Stream transforming

- Stream<R> cast<R>();
- Stream<S> map<S>(S Function(T event) convert);
- Stream<T> skip(int count);
- Stream<T> skipWhile(bool Function(T element) test);
- Stream<T> take(int count);
- Stream<T> where(bool Function(T event) test);
- Stream<T> distinct([bool Function(T previous, T next)? equals]);

listen()

- Returns StreamSubscription<T> listen(void Function(T event)? onData, {Function? onError, void Function()? onDone, bool? cancelOnError});
- You can cancel subscription via StreamSubscription

Example

```
var subscription = Stream<int>
    .periodic(const Duration(milliseconds: 100), (int event) ⇒ event)
    .map((event) ⇒ event * 100) // transform each element
    .where((event) ⇒ event > 500) // filter elements ⩽ 500
    .listen(
       (event) {
         print(event);     // handle elements
       },
       onError: (e) {
         print(e);        // handle an error
       },
    );
```

Libraries

pubspec.yaml

1. Add dependency name into pubspec.yaml

```
dependencies:
  http: any
```

2. `flutter pub get`

3. Use it!

pubspec.yaml

1. Add dependency name into pubspec.yaml

```
dependencies:
  http: any
```

2. `flutter pub get`

3. Use it!

pub.dev

- Common repository for all dart/flutter libraries
- Many criterias: pub points, popularity, likes, verified publisher, etc
- Easy to use
- Easy to publish

https://pub.dev/packages/http
https://pub.dev/packages/flip_card

Versioning

```yaml
environment:
  sdk: "⩾2.12.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

  http: any
  flip_card: any

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^1.0.0
```

https://dart.dev/tools/pub/versioning

Versioning

```
environment:
  sdk: "⩾2.12.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

  http: any
  flip_card: any

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^1.0.0
```

https://dart.dev/tools/pub/versioning

Versioning

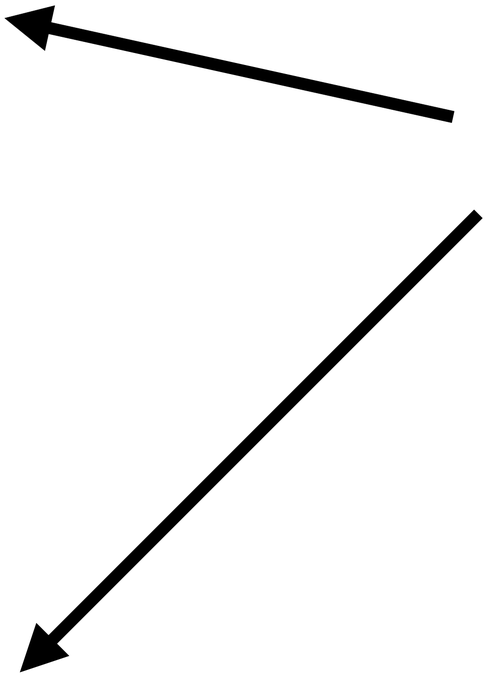
```
environment:
  sdk: "⩾2.12.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

  http: any
  flip_card: any

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^0.2.0
```



Version Constraints

based on semver

```
// caret syntax  -  "⩾1.0.0 <2.0.0"
```

https://dart.dev/tools/pub/versioning

Yandex

Thank you for attention

Sergey Koltsov, Yandex.Pro Team Lead

ringov@yandex-team.ru