# Introduction to Dart

This page provides a brief introduction to the Dart language through samples of its main features.

To learn more about the Dart language, visit the in-depth, individual topic pages listed under **Language** in the left side menu.

For coverage of Dart's core libraries, check out the core library documentation. You can also check out the Dart cheatsheet, for a more interactive introduction.

## Hello World

Every app requires the top-level `main()` function, where execution starts. Functions that don't explicitly return a value have the `void` return type. To display text on the console, you can use the top-level `print()` function:

```dart
void main() {
  print('Hello, World!');
}
```

Read more about the `main()` function in Dart, including optional parameters for command-line arguments.

## Variables

Even in type-safe Dart code, you can declare most variables without explicitly specifying their type using `var`. Thanks to type inference, these variables' types are determined by their initial values:

```dart
var name = 'Voyager I';
var year = 1977;
var antennaDiameter = 3.7;
var flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
var image = {
  'tags': ['saturn'],
  'url': '//path/to/saturn.jpg'
};
```

Read more about variables in Dart, including default values, the `final` and `const` keywords, and static types.

## Control flow statements

Dart supports the usual control flow statements:

```dart
if (year >= 2001) {
  print('21st century');
} else if (year >= 1901) {
  print('20th century');
}

for (final object in flybyObjects) {
  print(object);
}

for (int month = 1; month <= 12; month++) {
  print(month);
}

while (year < 2016) {
  year += 1;
}
```

Read more about control flow statements in Dart, including break and continue, switch and case, and assert.

## Functions

We recommend specifying the types of each function's arguments and return value:

```dart
int fibonacci(int n) {
  if (n == 0 || n == 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

var result = fibonacci(20);
```

A shorthand `=>` (*arrow*) syntax is handy for functions that contain a single statement. This syntax is especially useful when passing anonymous functions as arguments:

```dart
flybyObjects.where((name) => name.contains('turn')).forEach(print);
```

Besides showing an anonymous function (the argument to `where()`), this code shows that you can use a function as an argument: the top-level `print()` function is an argument to `forEach()`.

Read more about functions in Dart, including optional parameters, default parameter values, and lexical scope.

## Comments

Dart comments usually start with `//`.

```dart
// This is a normal, one-line comment.

/// This is a documentation comment, used to document libraries,
/// classes, and their members. Tools like IDEs and dartdoc treat
/// doc comments specially.

/* Comments like these are also supported. */
```

Read more about comments in Dart, including how the documentation tooling works.

## Imports

To access APIs defined in other libraries, use `import`.

```dart
// Importing core libraries
import 'dart:math';

// Importing libraries from external packages
import 'package:test/test.dart';

// Importing files
import 'path/to/my_other_file.dart';
```

Read more about libraries and visibility in Dart, including library prefixes, `show` and `hide`, and lazy loading through the `deferred` keyword.

## Classes

Here's an example of a class with three properties, two constructors, and a method. One of the properties can't be set directly, so it's defined using a getter method (instead of a variable). The method uses string interpolation to print variables' string equivalents inside of string literals.

```dart
class Spacecraft {
  String name;
  DateTime? launchDate;

  // Read-only non-final property
  int? get launchYear => launchDate?.year;

  // Constructor, with syntactic sugar for assignment to members.
  Spacecraft(this.name, this.launchDate) {
    // Initialization code goes here.
  }

  // Named constructor that forwards to the default one.
  Spacecraft.unlaunched(String name) : this(name, null);

  // Method.
  void describe() {
    print('Spacecraft: $name');
    // Type promotion doesn't work on getters.
    var launchDate = this.launchDate;
    if (launchDate != null) {
      int years = DateTime.now().difference(launchDate).inDays ~/ 365;
      print('Launched: $launchYear ($years years ago)');
    } else {
      print('Unlaunched');
    }
  }
}
```

Read more about strings, including string interpolation, literals, expressions, and the `toString()` method.

You might use the `Spacecraft` class like this:

```dart
var voyager = Spacecraft('Voyager I', DateTime(1977, 9, 5));
voyager.describe();

var voyager3 = Spacecraft.unlaunched('Voyager III');
voyager3.describe();
```

Read more about classes in Dart, including initializer lists, optional `new` and `const`, redirecting constructors, `factory` constructors, getters, setters, and much more.

## Enums

Enums are a way of enumerating a predefined set of values or instances in a way which ensures that there cannot be any other instances of that type.

Here is an example of a simple `enum` that defines a simple list of predefined planet types:

```dart
enum PlanetType { terrestrial, gas, ice }
```

Here is an example of an enhanced enum declaration of a class describing planets, with a defined set of constant instances, namely the planets of our own solar system.

```dart
/// Enum that enumerates the different planets in our solar system
/// and some of their properties.
enum Planet {
  mercury(planetType: PlanetType.terrestrial, moons: 0, hasRings: false),
  venus(planetType: PlanetType.terrestrial, moons: 0, hasRings: false),
  // ···
  uranus(planetType: PlanetType.ice, moons: 27, hasRings: true),
  neptune(planetType: PlanetType.ice, moons: 14, hasRings: true);

  /// A constant generating constructor
  const Planet(
      {required this.planetType, required this.moons, required this.hasRings});

  /// All instance variables are final
  final PlanetType planetType;
  final int moons;
  final bool hasRings;

  /// Enhanced enums support getters and other methods
  bool get isGiant =>
      planetType == PlanetType.gas || planetType == PlanetType.ice;
}
```

You might use the `Planet` enum like this:

```dart
final yourPlanet = Planet.earth;

if (!yourPlanet.isGiant) {
  print('Your planet is not a "giant planet".');
}
```

Read more about enums in Dart, including enhanced enum requirements, automatically introduced properties, accessing enumerated value names, switch statement support, and much more.

## Inheritance

Dart has single inheritance.

```dart
class Orbiter extends Spacecraft {
  double altitude;

  Orbiter(super.name, DateTime super.launchDate, this.altitude);
}
```

Read more about extending classes, the optional `@override` annotation, and more.

## Mixins

Mixins are a way of reusing code in multiple class hierarchies. The following is a mixin declaration:

```dart
mixin Piloted {
  int astronauts = 1;

  void describeCrew() {
    print('Number of astronauts: $astronauts');
  }
}
```

To add a mixin's capabilities to a class, just extend the class with the mixin.

```dart
class PilotedCraft extends Spacecraft with Piloted {
  // ···
}
```

`PilotedCraft` now has the `astronauts` field as well as the `describeCrew()` method.

Read more about mixins.

## Interfaces and abstract classes

All classes implicitly define an interface. Therefore, you can implement any class.

```dart
class MockSpaceship implements Spacecraft {
  // ···
}
```

Read more about implicit interfaces, or about the explicit `interface` keyword.

You can create an abstract class to be extended (or implemented) by a concrete class. Abstract classes can contain abstract methods (with empty bodies).

```dart
abstract class Describable {
  void describe();

  void describeWithEmphasis() {
    print('=========');
    describe();
    print('=========');
  }
}
```

Any class extending `Describable` has the `describeWithEmphasis()` method, which calls the extender's implementation of `describe()`.

Read more about abstract classes and methods.

## Async

Avoid callback hell and make your code much more readable by using `async` and `await`.

```dart
const oneSecond = Duration(seconds: 1);
// ···
Future<void> printWithDelay(String message) async {
  await Future.delayed(oneSecond);
  print(message);
}
```

The method above is equivalent to:

```dart
Future<void> printWithDelay(String message) {
  return Future.delayed(oneSecond).then((_) {
    print(message);
  });
}
```

As the next example shows, `async` and `await` help make asynchronous code easy to read.

```dart
Future<void> createDescriptions(Iterable<String> objects) async {
  for (final object in objects) {
    try {
      var file = File('$object.txt');
      if (await file.exists()) {
        var modified = await file.lastModified();
        print(
            'File for $object already exists. It was modified on $modified.');
        continue;
      }
      await file.create();
      await file.writeAsString('Start describing $object in this file.');
    } on IOException catch (e) {
      print('Cannot create description for $object: $e');
    }
  }
}
```

You can also use `async*`, which gives you a nice, readable way to build streams.

```dart
Stream<String> report(Spacecraft craft, Iterable<String> objects) async* {
  for (final object in objects) {
    await Future.delayed(oneSecond);
    yield '${craft.name} flies by $object';
  }
}
```

Read more about asynchrony support, including `async` functions, `Future`, `Stream`, and the asynchronous loop (`await for`).

## Exceptions

To raise an exception, use `throw`:

```dart
if (astronauts == 0) {
  throw StateError('No astronauts.');
}
```

To catch an exception, use a `try` statement with `on` or `catch` (or both):

```dart
Future<void> describeFlybyObjects(List<String> flybyObjects) async {
  try {
    for (final object in flybyObjects) {
      var description = await File('$object.txt').readAsString();
      print(description);
    }
  } on IOException catch (e) {
    print('Could not describe object: $e');
  } finally {
    flybyObjects.clear();
  }
}
```

Note that the code above is asynchronous; `try` works for both synchronous code and code in an `async` function.

Read more about exceptions, including stack traces, `rethrow`, and the difference between `Error` and `Exception`.

## Important concepts

As you continue to learn about the Dart language, keep these facts and concepts in mind:

- Everything you can place in a variable is an *object*, and every object is an instance of a *class*. Even numbers, functions, and `null` are objects. With the exception of `null` (if you enable sound null safety), all objects inherit from the `Object` class.

  > **Version note**
  >
  > Null safety was introduced in Dart 2.12. Using null safety requires a language version of at least 2.12.

- Although Dart is strongly typed, type annotations are optional because Dart can infer types. In `var number = 101`, `number` is inferred to be of type `int`.

- If you enable null safety, variables can't contain `null` unless you say they can. You can make a variable nullable by putting a question mark (`?`) at the end of its type. For example, a variable of type `int?` might be an integer, or it might be `null`. If you *know* that an expression never evaluates to `null` but Dart disagrees, you can add `!` to assert that it

isn't null (and to throw an exception if it is). An example: `int x = nullableButNotNullInt!`

- When you want to explicitly say that any type is allowed, use the type `Object?` (if you've enabled null safety), `Object`, or—if you must defer type checking until runtime—the special type `dynamic`.
- Dart supports generic types, like `List<int>` (a list of integers) or `List<Object>` (a list of objects of any type).
- Dart supports top-level functions (such as `main()`), as well as functions tied to a class or object (*static* and *instance methods*, respectively). You can also create functions within functions (*nested* or *local functions*).
- Similarly, Dart supports top-level *variables*, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as *fields* or *properties*.
- Unlike Java, Dart doesn't have the keywords `public`, `protected`, and `private`. If an identifier starts with an underscore (_), it's private to its library. For details, see Libraries and imports.
- *Identifiers* can start with a letter or underscore (_), followed by any combination of those characters plus digits.
- Dart has both *expressions* (which have runtime values) and *statements* (which don't). For example, the conditional expression `condition ? expr1 : expr2` has a value of `expr1` or `expr2`. Compare that to an if-else statement, which has no value. A statement often contains one or more expressions, but an expression can't directly contain a statement.
- Dart tools can report two kinds of problems: *warnings* and *errors*. Warnings are just indications that your code might not work, but they don't prevent your program from executing. Errors can be either compile-time or run-time. A compile-time error prevents the code from executing at all; a run-time error results in an exception being raised while the code executes.

## Additional resources

You can find more documentation and code samples in the core library documentation and the Dart API reference. This site's code follows the conventions in the Dart style guide.