Functions

Dart is a true object-oriented language, so even functions are objects and have a type, Function. This means that functions can be assigned to variables or passed as arguments to other functions. You can also call an instance of a Dart class as if it were a function. For details, see Callable objects.

Here's an example of implementing a function:

```dart
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

Although Effective Dart recommends type annotations for public APIs, the function still works if you omit the types:

```dart
isNoble(atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

For functions that contain just one expression, you can use a shorthand syntax:

```dart
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

The `=> expr` syntax is a shorthand for `{ return expr; }`. The `=>` notation is sometimes referred to as *arrow* syntax.

> ⓘ 提示
>
> Only *expressions* can appear between the arrow (`=>`) and the semicolon (`;`). Expressions evaluate to values. This means that you can't write a statement where Dart expects a value. For example, you could use a conditional expression but not an if statement. In the previous example, `_nobleGases[atomicNumber] != null;` returns a boolean value. The function then returns a boolean value that indicates whether the `atomicNumber` falls into the noble gas range.

Parameters

A function can have any number of *required positional* parameters. These can be followed either by *named* parameters or by *optional positional* parameters (but not both).

> ⓘ 提示
>
> Some APIs—notably Flutter widget constructors—use only named parameters, even for parameters that are mandatory. See the next section for details.

You can use trailing commas when you pass arguments to a function or when you define function parameters.

Named parameters

Named parameters are optional unless they're explicitly marked as `required`.

When defining a function, use `{param1, param2, …}` to specify named parameters. If you don't provide a default value or mark a named parameter as `required`, their types must be nullable as their default value will be `null`:

```dart
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool? bold, bool? hidden}) {...}
```

When calling a function, you can specify named arguments using *`paramName: value`*. For example:

```dart
enableFlags(bold: true, hidden: false);
```

To define a default value for a named parameter besides `null`, use `=` to specify a default value. The specified value must be a compile-time constant. For example:

```dart
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool bold = false, bool hidden = false}) {...}

// bold will be true; hidden will be false.
enableFlags(bold: true);
```

If you instead want a named parameter to be mandatory, requiring callers to provide a value for the parameter, annotate them with `required`:

```dart
const Scrollbar({super.key, required Widget child});
```

If someone tries to create a `Scrollbar` without specifying the `child` argument, then the analyzer reports an issue.

ⓘ 提示

A parameter marked as `required` can still be nullable:

```dart
const Scrollbar({super.key, required Widget? child});
```

You might want to place positional arguments first, but Dart doesn't require it. Dart allows named arguments to be placed anywhere in the argument list when it suits your API:

```dart
repeat(times: 2, () {
  ...
});
```

Optional positional parameters

Wrapping a set of function parameters in `[]` marks them as optional positional parameters. If you don't provide a default value, their types must be nullable as their default value will be `null`:

```dart
String say(String from, String msg, [String? device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Here's an example of calling this function without the optional parameter:

```dart
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

And here's an example of calling this function with the third parameter:

```dart
assert(say('Bob', 'Howdy', 'smoke signal') ==
    'Bob says Howdy with a smoke signal');
```

To define a default value for an optional positional parameter besides `null`, use `=` to specify a default value. The specified value must be a compile-time constant. For example:

```dart
String say(String from, String msg, [String device = 'carrier pigeon']) {
  var result = '$from says $msg with a $device';
  return result;
}

assert(say('Bob', 'Howdy') == 'Bob says Howdy with a carrier pigeon');
```

The main() function

Every app must have a top-level `main()` function, which serves as the entrypoint to the app. The `main()` function returns `void` and has an optional `List<String>` parameter for arguments.

Here's a simple `main()` function:

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

Here's an example of the `main()` function for a command-line app that takes arguments:

args.dart

```dart
// Run the app like this: dart run args.dart 1 test
void main(List<String> arguments) {
  print(arguments);

  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

You can use the args library to define and parse command-line arguments.

Functions as first-class objects

You can pass a function as a parameter to another function. For example:

```dart
void printElement(int element) {
  print(element);
}

var list = [1, 2, 3];

// Pass printElement as a parameter.
list.forEach(printElement);
```

You can also assign a function to a variable, such as:

```dart
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') == '!!! HELLO !!!');
```

This example uses an anonymous function. More about those in the next section.

Function types

You can specify the type of a function, which is known as a *function type*. A function type is obtained from a function declaration header by replacing the function name by the keyword `Function`. Moreover, you are allowed to omit the names of positional parameters, but the names of named parameters can't be omitted. For example:

```dart
void greet(String name, {String greeting = 'Hello'}) =>
    print('$greeting $name!');

// Store `greet` in a variable and call it.
void Function(String, {String greeting}) g = greet;
g('Dash', greeting: 'Howdy');
```

ⓘ 提示

In Dart, functions are first-class objects, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

You can use a `typedef` declaration to explicitly name function types, which can be useful for clarity and reusability.

Anonymous functions

Though you name most functions, such as `main()` or `printElement()`. you can also create functions without names. These functions are called *anonymous functions*, *lambdas*, or *closures*.

An anonymous function resembles a named function as it has:

- Zero or more parameters, comma-separated
- Optional type annotations between parentheses.

The following code block contains the function's body:

```dart
([[Type] param1[, ...]]) {
  codeBlock;
}
```

The following example defines an anonymous function with an untyped parameter, `item`. The anonymous function passes it to the `map` function. The `map` function, invoked for each item in the list, converts each string to uppercase. Then, the anonymous function passed to `forEach`, prints each converted string with its length.

```dart
const list = ['apples', 'bananas', 'oranges'];

var uppercaseList = list.map((item) {
  return item.toUpperCase();
}).toList();
// Convert to list after mapping

for (var item in uppercaseList) {
  print('$item: ${item.length}');
}
```

Click **Run** to execute the code.

If the function contains only a single expression or return statement, you can shorten it using arrow notation. Paste the following line into DartPad and click **Run** to verify that it is functionally equivalent.

```dart
var uppercaseList = list.map((item) => item.toUpperCase()).toList();
uppercaseList.forEach((item) => print('$item: ${item.length}'));
```

Lexical scope

Dart determines the scope of variables based on the layout of its code. A programming language with this feature is termed a lexically scoped language. You can "follow the curly braces outwards" to see if a variable is in scope.

Example: A series of nested functions with variables at each scope level:

```dart
bool topLevel = true;

void main() {
  var insideMain = true;

  void myFunction() {
    var insideFunction = true;

    void nestedFunction() {
      var insideNestedFunction = true;

      assert(topLevel);
      assert(insideMain);
      assert(insideFunction);
      assert(insideNestedFunction);
    }
  }
}
```

The `nestedFunction()` method can use variables from every level, all the way up to the top level.

Lexical closures

A function object that can access variables in its lexical scope when the function sits outside that scope is called a *closure*.

Functions can close over variables defined in surrounding scopes. In the following example, `makeAdder()` captures the variable `addBy`. Wherever the returned function goes, it remembers `addBy`.

```dart
/// Returns a function that adds [addBy] to the
/// function's argument.
Function makeAdder(int addBy) {
  return (int i) => addBy + i;
}

void main() {
  // Create a function that adds 2.
  var add2 = makeAdder(2);

  // Create a function that adds 4.
  var add4 = makeAdder(4);

  assert(add2(3) == 5);
  assert(add4(3) == 7);
}
```

Tear-offs

When you refer to a function, method, or named constructor without parentheses, Dart creates a *tear-off*. This is a closure that takes the same parameters as the function and invokes the underlying function when you call it. If your code needs a closure that invokes a named function with the same parameters as the closure accepts, don't wrap the call in a lambda. Use a tear-off.

```dart
var charCodes = [68, 97, 114, 116];
var buffer = StringBuffer();
```

good

```dart
// Function tear-off
charCodes.forEach(print);

// Method tear-off
charCodes.forEach(buffer.write);
```

bad

```dart
// Function lambda
charCodes.forEach((code) {
  print(code);
});

// Method lambda
charCodes.forEach((code) {
  buffer.write(code);
});
```

Testing functions for equality

Here's an example of testing top-level functions, static methods, and instance methods for equality:

```dart
void foo() {} // A top-level function

class A {
  static void bar() {} // A static method
  void baz() {} // An instance method
}

void main() {
  Function x;

  // Comparing top-level functions.
  x = foo;
  assert(foo == x);

  // Comparing static methods.
  x = A.bar;
  assert(A.bar == x);

  // Comparing instance methods.
  var v = A(); // Instance #1 of A
  var w = A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // These closures refer to the same instance (#2),
  // so they're equal.
  assert(y.baz == x);

  // These closures refer to different instances,
  // so they're unequal.
  assert(v.baz != w.baz);
}
```

Return values

All functions return a value. If no return value is specified, the statement `return null;` is implicitly appended to the function body.

```dart
foo() {}

assert(foo() == null);
```

To return multiple values in a function, aggregate the values in a record.

```dart
(String, int) foo() {
  return ('something', 42);
}
```

Generators

When you need to lazily produce a sequence of values, consider using a *generator function*. Dart has built-in support for two kinds of generator functions:

- **Synchronous** generator: Returns an `Iterable` object.
- **Asynchronous** generator: Returns a `Stream` object.

To implement a **synchronous** generator function, mark the function body as `sync*`, and use `yield` statements to deliver values:

```dart
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}
```

To implement an **asynchronous** generator function, mark the function body as `async*`, and use `yield` statements to deliver values:

```dart
Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}
```

If your generator is recursive, you can improve its performance by using `yield*`:

```dart
Iterable<int> naturalsDownFrom(int n) sync* {
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n - 1);
  }
}
```

External functions

An external function is a function whose body is implemented separately from its declaration. Include the `external` keyword before a function declaration, like so:

```dart
external void someFunc(int i);
```

An external function's implementation can come from another Dart library, or, more commonly, from another language. In interop contexts, `external` introduces type information for foreign functions or values, making them usable in Dart. Implementation and usage is heavily platform specific, so check out the interop docs on, for example, C or JavaScript to learn more.

External functions can be top-level functions, instance methods, getters or setters, or non-redirecting constructors. An instance variable can be `external` too, which is equivalent to an external getter and (if the variable is not `final`) an external setter.