

A tour of the Dart language

This page shows you how to use each major Dart feature, from variables and operators to classes and libraries, with the assumption that you already know how to program in another language. For a briefer, less complete introduction to the language, see the [language samples page](#).

To learn more about Dart's core libraries, see the [library tour](#). Whenever you want more details about a language feature, consult the [Dart language specification](#).

Note: You can play with most of Dart's language features using DartPad ([learn more](#)). [Open DartPad](#).

This page uses embedded DartPads to display some of the examples. If you see empty boxes instead of DartPads, go to the [DartPad troubleshooting page](#).

A basic Dart program

The following code uses many of Dart's most basic features:

```
// Define a function.
void printInteger(int aNumber) {
  print('The number is $aNumber.');
```

// Print to console.

```
}

// This is where the app starts executing.
void main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}
```

Here's what this program uses that applies to all (or almost all) Dart apps:

// This is a comment.

A single-line comment. Dart also supports multi-line and document comments. For details, see [Comments](#).

`void`

A special type that indicates a value that's never used. Functions like `printInteger()` and `main()` that don't explicitly return a value have the `void` return type.

`int`

Another type, indicating an integer. Some additional [built-in types](#) are `String`, `List`, and `bool`.

`42`

A number literal. Number literals are a kind of compile-time constant.

`print()`

A handy way to display output.

`'...'` (or `"..."`)

A string literal.

`$variableName` (or `${expression}`)

String interpolation: including a variable or expression's string equivalent inside of a string literal. For more information, see [Strings](#).

`main()`

The special, *required*, top-level function where app execution starts. For more information, see [The main\(\) function](#).

`var`

A way to declare a variable without specifying its type. The type of this variable (`int`) is determined by its initial value (`42`).

Note: This site's code follows the conventions in the [Dart style guide](#).

Important concepts

As you 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 the code above, `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 visibility](#).
- 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.

Keywords

The following table lists the words that the Dart language treats specially.

abstract ²	else	import ²	show ¹
as ²	enum	in	static ²
assert	export ²	interface ²	super
async ¹	extends	is	switch
await ³	extension ²	late ²	sync ¹

break	external ²	library ²	this
case	factory ²	mixin ²	throw
catch	false	new	true
class	final	null	try
const	finally	on ¹	typedef ²
continue	for	operator ²	var
covariant ²	Function ²	part ²	void
default	get ²	required ²	while
deferred ²	hide ¹	rethrow	with
do	if	return	yield ³
dynamic ²	implements ²	set ²	

Avoid using these words as identifiers. However, if necessary, the keywords marked with superscripts can be identifiers:

- Words with the superscript **1** are **contextual keywords**, which have meaning only in specific places. They're valid identifiers everywhere.
- Words with the superscript **2** are **built-in identifiers**. These keywords are valid identifiers in most places, but they can't be used as class or type names, or as import prefixes.
- Words with the superscript **3** are limited reserved words related to [asynchrony support](#). You can't use `await` or `yield` as an identifier in any function body marked with `async`, `async*`, or `sync*`.

All other words in the table are **reserved words**, which can't be identifiers.

Variables

Here's an example of creating a variable and initializing it:

```
var name = 'Bob';
```

Variables store references. The variable called `name` contains a reference to a `String` object with a value of "Bob".

The type of the `name` variable is inferred to be `String`, but you can change that type by specifying it. If an object isn't restricted to a single type, specify the `Object` type (or `dynamic` if necessary).

```
Object name = 'Bob';
```

Another option is to explicitly declare the type that would be inferred:

```
String name = 'Bob';
```

Note: This page follows the [style guide recommendation](#) of using `var`, rather than type annotations, for local variables.

Default value

Uninitialized variables that have a nullable type have an initial value of `null`. (If you haven't opted into [null safety](#), then every variable has a nullable type.) Even variables with numeric types are initially null, because numbers—like everything else in Dart—are objects.

```
int? lineCount;  
assert(lineCount == null);
```

Note: Production code ignores the `assert()` call. During development, on the other hand, `assert(condition)` throws an exception if *condition* is false. For details, see [Assert](#).

If you enable null safety, then you must initialize the values of non-nullable variables before you use them:

```
int lineCount = 0;
```

You don't have to initialize a local variable where it's declared, but you do need to assign it a value before it's used. For example, the following code is valid because Dart can detect that `lineCount` is non-null by the time it's passed to `print()`:

```
int lineCount;  
  
if (weLikeToCount) {  
  lineCount = countLines();  
} else {  
  lineCount = 0;  
}  
  
print(lineCount);
```

Top-level and class variables are lazily initialized; the initialization code runs the first time the variable is used.

Late variables

Dart 2.12 added the `late` modifier, which has two use cases:

- Declaring a non-nullable variable that's initialized after its declaration.
- Lazily initializing a variable.

Often Dart's control flow analysis can detect when a non-nullable variable is set to a non-null value before it's used, but sometimes analysis fails. Two common cases are top-level variables and instance variables: Dart often can't determine whether they're set, so it doesn't try.

If you're sure that a variable is set before it's used, but Dart disagrees, you can fix the error by marking the variable as `late`:

```
late String description;  
  
void main() {  
  description = 'Feijooda!';  
  print(description);  
}
```

Warning: If you fail to initialize a `late` variable, a runtime error occurs when the variable is used.

When you mark a variable as `late` but initialize it at its declaration, then the initializer runs the first time the variable is used. This lazy initialization is handy in a couple of cases:

- The variable might not be needed, and initializing it is costly.
- You're initializing an instance variable, and its initializer needs access to `this`.

In the following example, if the `temperature` variable is never used, then the expensive `readThermometer()` function is never called:

```
// This is the program's only call to readThermometer().  
late String temperature = readThermometer(); // Lazily initialized.
```

Final and const

If you never intend to change a variable, use `final` or `const`, either instead of `var` or in addition to a type. A final variable can be set only once; a const variable is a compile-time constant. (Const variables are implicitly final.)

Note: [Instance variables](#) can be `final` but not `const`.

Here's an example of creating and setting a `final` variable:

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

You can't change the value of a `final` variable:

```
x static analysis: error/warning  
name = 'Alice'; // Error: a final variable can only be set once.
```

Use `const` for variables that you want to be **compile-time constants**. If the const variable is at the class level, mark it `static const`. Where you declare the variable, set the value to a compile-time constant such as a number or string literal, a const variable, or the result of an arithmetic operation on constant numbers:

```
const bar = 1000000; // Unit of pressure (dynes/cm2)  
const double atm = 1.01325 * bar; // Standard atmosphere
```

The `const` keyword isn't just for declaring constant variables. You can also use it to create constant *values*, as well as to declare constructors that *create* constant values. Any variable can have a constant value.

```
var foo = const [];  
final bar = const [];  
const baz = []; // Equivalent to 'const []'
```

You can omit `const` from the initializing expression of a `const` declaration, like for `baz` above. For details, see [DON'T use const redundantly](#).

You can change the value of a non-final, non-const variable, even if it used to have a `const` value:

```
foo = [1, 2, 3]; // Was const []
```

You can't change the value of a `const` variable:

```
x static analysis: error/warning
baz = [42]; // Error: Constant variables can't be assigned a value.
```

You can define constants that use [type checks and casts](#) (`is` and `as`), [collection if](#), and [spread operators](#) (`...` and `...?`):

```
const Object i = 3; // Where i is a const Object with an int value...
const list = [i as int]; // Use a typecast.
const map = {if (i is int) i: 'int'}; // Use is and collection if.
const set = {if (list is List<int>) ...list}; // ...and a spread.
```

Note: Although a `final` object cannot be modified, its fields can be changed. In comparison, a `const` object and its fields cannot be changed: they're *immutable*.

For more information on using `const` to create constant values, see [Lists](#), [Maps](#), and [Classes](#).

Built-in types

The Dart language has special support for the following:

- [Numbers](#) (`int`, `double`)
- [Strings](#) (`String`)
- [Booleans](#) (`bool`)
- [Lists](#) (`List`, also known as *arrays*)
- [Sets](#) (`Set`)
- [Maps](#) (`Map`)
- [Runes](#) (`Runes`; often replaced by the `characters` API)
- [Symbols](#) (`Symbol`)
- The value `null` (`Null`)

This support includes the ability to create objects using literals. For example, `'this is a string'` is a string literal, and `true` is a boolean literal.

Because every variable in Dart refers to an object—an instance of a *class*—you can usually use *constructors* to initialize variables. Some of the built-in types have their own constructors. For example, you can use the `Map()` constructor to create a map.

Some other types also have special roles in the Dart language:

- **Object**: The superclass of all Dart classes except `Null`.
- **Enum**: The superclass of all enums.
- **Future** and **Stream**: Used in [asynchrony support](#).
- **Iterable**: Used in [for-in loops](#) and in synchronous [generator functions](#).
- **Never**: Indicates that an expression can never successfully finish evaluating. Most often used for functions that always throw an exception.
- **dynamic**: Indicates that you want to disable static checking. Usually you should use `Object` or `Object?` instead.
- **void**: Indicates that a value is never used. Often used as a return type.

The `Object`, `Object?`, `Null`, and `Never` classes have special roles in the class hierarchy, as described in the [top-and-bottom](#) section of [Understanding null safety](#).

Numbers

Dart numbers come in two flavors:

[int](#)

Integer values no larger than 64 bits, [depending on the platform](#). On native platforms, values can be from -2^{63} to $2^{63} - 1$. On the web, integer values are represented as JavaScript numbers (64-bit floating-point values with no fractional part) and can be from -2^{53} to $2^{53} - 1$.

[double](#)

64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard.

Both `int` and `double` are subtypes of `num`. The `num` type includes basic operators such as `+`, `-`, `/`, and `*`, and is also where you'll find `abs()`, `ceil()`, and `floor()`, among other methods. (Bitwise operators, such as `>>`, are defined in the `int` class.) If `num` and its subtypes don't have what you're looking for, the [dart:math](#) library might.

Integers are numbers without a decimal point. Here are some examples of defining integer literals:

```
var x = 1;
var hex = 0xDEADBEEF;
```

If a number includes a decimal, it is a double. Here are some examples of defining double literals:

```
var y = 1.1;
var exponents = 1.42e5;
```

You can also declare a variable as a `num`. If you do this, the variable can have both integer and double values.

```
num x = 1; // x can have both int and double values
x += 2.5;
```

Integer literals are automatically converted to doubles when necessary:

```
double z = 1; // Equivalent to double z = 1.0.
```

Here's how you turn a string into a number, or vice versa:

```
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

The `int` type specifies the traditional bitwise shift (`<<`, `>>`, `>>>`), complement (`~`), AND (`&`), OR (`|`), and XOR (`^`) operators, which are useful for manipulating and masking flags in bit fields. For example:

```
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 | 4) == 7); // 0011 | 0100 == 0111
assert((3 & 4) == 0); // 0011 & 0100 == 0000
```

For more examples, see the [bitwise and shift operator](#) section.

Literal numbers are compile-time constants. Many arithmetic expressions are also compile-time constants, as long as their operands are compile-time constants that evaluate to numbers.

```
const msPerSecond = 1000;
const secondsUntilRetry = 5;
const msUntilRetry = secondsUntilRetry * msPerSecond;
```

For more information, see [Numbers in Dart](#).

Strings

A Dart string (`String` object) holds a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

You can put the value of an expression inside a string by using `${expression}`. If the expression is an identifier, you can skip the `{}`. To get the string corresponding to an object, Dart calls the object's `toString()` method.

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
      'Dart has string interpolation, '
      'which is very handy.');
```

```
assert('That deserves all caps. '
      '${s.toUpperCase()} is very handy!' ==
      'That deserves all caps. '
      'STRING INTERPOLATION is very handy!');
```

Note: The `==` operator tests whether two objects are equivalent. Two strings are equivalent if they contain the same sequence of code units.

You can concatenate strings using adjacent string literals or the `+` operator:

```
var s1 = 'String '
      'concatenation'
      " works even over line breaks.";
assert(s1 ==
      'String concatenation works even over '
      'line breaks.');
```

```
var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

Another way to create a multi-line string: use a triple quote with either single or double quotation marks:

```
var s1 = '''
You can create
multi-line strings like this one.
''';

var s2 = """This is also a
multi-line string.""";
```


You can create a “raw” string by prefixing it with `r`:

```
var s = r'In a raw string, not even \n gets special treatment.';
```

See [Runes and grapheme clusters](#) for details on how to express Unicode characters in a string.

Literal strings are compile-time constants, as long as any interpolated expression is a compile-time constant that evaluates to null or a numeric, string, or boolean value.

```
// These work in a const string.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// These do NOT work in a const string.
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];

const validConstString = '$aConstNum $aConstBool $aConstString';
// const invalidConstString = '$aNum $aBool $aString $aConstList';
```

For more information on using strings, see [Strings and regular expressions](#).

Booleans

To represent boolean values, Dart has a type named `bool`. Only two objects have type `bool`: the boolean literals `true` and `false`, which are both compile-time constants.

Dart’s type safety means that you can’t use code like `if (nonbooleanValue)` or `assert (nonbooleanValue)`. Instead, explicitly check for values, like this:

```
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty);

// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Check for null.
var unicorn;
assert(unicorn == null);

// Check for NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

Lists

Perhaps the most common collection in nearly every programming language is the *array*, or ordered group of objects. In Dart, arrays are [List](#) objects, so most people just call them *lists*.

Dart list literals are denoted by a comma separated list of expressions or values, enclosed in square brackets (`[]`). Here’s a simple Dart list:

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

Note: Dart infers that `list` has type `List<int>`. If you try to add non-integer objects to this list, the analyzer or runtime raises an error. For more information, read about [type inference](#).

You can add a comma after the last item in a Dart collection literal. This *trailing comma* doesn't affect the collection, but it can help prevent copy-paste errors.

```
var list = [  
  'Car',  
  'Boat',  
  'Plane',  
];
```

Lists use zero-based indexing, where 0 is the index of the first value and `list.length - 1` is the index of the last value. You can get a list's length using the `.length` property and access a list's values using the subscript operator (`[]`):

```
var list = [1, 2, 3];  
assert(list.length == 3);  
assert(list[1] == 2);  
  
list[1] = 1;  
assert(list[1] == 1);
```

To create a list that's a compile-time constant, add `const` before the list literal:

```
var constantList = const [1, 2, 3];  
// constantList[1] = 1; // This line will cause an error.
```

Dart supports the **spread operator** (`...`) and the **null-aware spread operator** (`...?`), which provide a concise way to insert multiple values into a collection.

For example, you can use the spread operator (`...`) to insert all the values of a list into another list:

```
var list = [1, 2, 3];  
var list2 = [0, ...list];  
assert(list2.length == 4);
```

If the expression to the right of the spread operator might be null, you can avoid exceptions by using a null-aware spread operator (`...?`):

```
var list2 = [0, ...?list];  
assert(list2.length == 1);
```

For more details and examples of using the spread operator, see the [spread operator proposal](#).

Dart also offers **collection if** and **collection for**, which you can use to build collections using conditionals (`if`) and repetition (`for`).

Here's an example of using **collection if** to create a list with three or four items in it:

```
var nav = ['Home', 'Furniture', 'Plants', if (promoActive) 'Outlet'];
```

Here's an example of using **collection for** to manipulate the items of a list before adding them to another list:

```
var listOfInts = [1, 2, 3];
var listOfStrings = ['#0', for (var i in listOfInts) '#$i'];
assert(listOfStrings[1] == '#1');
```

For more details and examples of using collection `if` and `for`, see the [control flow collections proposal](#).

The List type has many handy methods for manipulating lists. For more information about lists, see [Generics](#) and [Collections](#).

Sets

A set in Dart is an unordered collection of unique items. Dart support for sets is provided by set literals and the [Set](#) type.

Here is a simple Dart set, created using a set literal:

```
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
```

i Note: Dart infers that `halogens` has the type `Set<String>`. If you try to add the wrong type of value to the set, the analyzer or runtime raises an error. For more information, read about [type inference](#).

To create an empty set, use `{}` preceded by a type argument, or assign `{}` to a variable of type `Set`:

```
var names = <String>{};
// Set<String> names = {}; // This works, too.
// var names = {}; // Creates a map, not a set.
```

i Set or map? The syntax for map literals is similar to that for set literals. Because map literals came first, `{}` defaults to the `Map` type. If you forget the type annotation on `{}` or the variable it's assigned to, then Dart creates an object of type `Map<dynamic, dynamic>`.

Add items to an existing set using the `add()` or `addAll()` methods:

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
```

Use `.length` to get the number of items in the set:

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
assert(elements.length == 5);
```

To create a set that's a compile-time constant, add `const` before the set literal:

```
final constantSet = const {
  'fluorine',
  'chlorine',
  'bromine',
  'iodine',
  'astatine',
};
// constantSet.add('helium'); // This line will cause an error.
```

Sets support spread operators (`...` and `...?`) and collection `if` and `for`, just like lists do. For more information, see the [list spread operator](#) and [list collection operator](#) discussions.

For more information about sets, see [Generics](#) and [Sets](#).

Maps

In general, a map is an object that associates keys and values. Both keys and values can be any type of object. Each *key* occurs only once, but you can use the same *value* multiple times. Dart support for maps is provided by map literals and the [Map](#) type.

Here are a couple of simple Dart maps, created using map literals:

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings'
};

var nobleGases = {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};
```

Note: Dart infers that `gifts` has the type `Map<String, String>` and `nobleGases` has the type `Map<int, String>`. If you try to add the wrong type of value to either map, the analyzer or runtime raises an error. For more information, read about [type inference](#).

You can create the same objects using a `Map` constructor:

```
var gifts = Map<String, String>();
gifts['first'] = 'partridge';
gifts['second'] = 'turtledoves';
gifts['fifth'] = 'golden rings';

var nobleGases = Map<int, String>();
nobleGases[2] = 'helium';
nobleGases[10] = 'neon';
nobleGases[18] = 'argon';
```

Note: If you come from a language like C# or Java, you might expect to see `new Map()` instead of just `Map()`. In Dart, the `new` keyword is optional. For details, see [Using constructors](#).

Add a new key-value pair to an existing map using the subscript assignment operator (`[]`):

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Add a key-value pair
```

Retrieve a value from a map using the subscript operator (`[]`):

```
var gifts = {'first': 'partridge'};
assert(gifts['first'] == 'partridge');
```

If you look for a key that isn't in a map, you get `null` in return:

```
var gifts = {'first': 'partridge'};
assert(gifts['fifth'] == null);
```

Use `.length` to get the number of key-value pairs in the map:

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds';
assert(gifts.length == 2);
```

To create a map that's a compile-time constant, add `const` before the map literal:

```
final constantMap = const {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};

// constantMap[2] = 'Helium'; // This line will cause an error.
```

Maps support spread operators (`...` and `...?`) and collection `if` and `for`, just like lists do. For details and examples, see the [spread operator proposal](#) and the [control flow collections proposal](#).

For more information about maps, see the [generics](#) section and the library tour's coverage of the [Maps API](#).

Runes and grapheme clusters

In Dart, [runes](#) expose the Unicode code points of a string. You can use the [characters package](#) to view or manipulate user-perceived characters, also known as [Unicode \(extended\) grapheme clusters](#).

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world's writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing Unicode code points within a string requires special syntax. The usual way to express a Unicode code point is `\uXXXX`, where `XXXX` is a 4-digit hexadecimal value. For example, the heart character (❤️) is `\u2665`. To specify more or less than 4 hex digits, place the value in curly brackets. For example, the laughing emoji (😂) is `\u{1f606}`.

If you need to read or write individual Unicode characters, use the `characters` getter defined on `String` by the `characters` package. The returned [Characters](#) object is the string as a sequence of grapheme clusters. Here's an example of using the `characters` API:

```
import 'package:characters/characters.dart';

void main() {
  var hi = 'Hi 𐀀';
  print(hi);
  print('The end of the string: ${hi.substring(hi.length - 1)}');
  print('The last character: ${hi.characters.last}');
}
```

The output, depending on your environment, looks something like this:

```
$ dart run bin/main.dart
Hi 𐀀
The end of the string: ???
The last character: 𐀀
```

For details on using the characters package to manipulate strings, see the [example](#) and [API reference](#) for the characters package.

Symbols

A [Symbol](#) object represents an operator or identifier declared in a Dart program. You might never need to use symbols, but they're invaluable for APIs that refer to identifiers by name, because minification changes identifier names but not identifier symbols.

To get the symbol for an identifier, use a symbol literal, which is just `#` followed by the identifier:

```
#radix
#bar
```

Symbol literals are compile-time constants.

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 classes](#).

Here's an example of implementing a function:

```
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:

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

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

```
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.

Note: Only an *expression*—not a *statement*—can appear between the arrow (`=>`) and the semicolon (`;`). For example, you can't put an [if statement](#) there, but you can use a [conditional expression](#).

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).

Note: 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`:

```
/// 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:

```
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:

```
/// 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`:

```
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.

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

```
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:

```
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`:

```
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:

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

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

```
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:

```
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:

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

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


```
// Run the app like this: dart 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:

```
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:

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

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

Anonymous functions

Most functions are named, such as `main()` or `printElement()`. You can also create a nameless function called an *anonymous function*, or sometimes a *lambda* or *closure*. You might assign an anonymous function to a variable so that, for example, you can add or remove it from a collection.

An anonymous function looks similar to a named function—zero or more parameters, separated by commas and optional type annotations, between parentheses.

The code block that follows contains the function's body:

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

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

```
const list = ['apples', 'bananas', 'oranges'];
list.map((item) {
  return item.toUpperCase();
}).forEach((item) {
  print('$item: ${item.length}');
});
```

Click **Run** to execute the code.

Dart Install SDK Format Reset ▶ Run ⋮

8
1
void main() {
2
 const list = ['apples', 'bananas', 'oranges'];
3
 list.map((item) {
4
 return item.toUpperCase();
5
 }).forEach((item) {
6
no issues

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.

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

Lexical scope

Dart is a lexically scoped language, which means that the scope of variables is determined statically, simply by the layout of the code. You can “follow the curly braces outwards” to see if a variable is in scope.

Here is an example of nested functions with variables at each scope level:

```
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);
    }
  }
}
```

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

Lexical closures

A *closure* is a function object that has access to variables in its lexical scope, even when the function is used outside of its original scope.

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`.

```

/// 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);
}

```

Testing functions for equality

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

```

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.

```

foo() {}


assert(foo() == null);

```

Operators

Dart supports the operators shown in the following table. The table shows Dart’s operator associativity and [operator precedence](#) from highest to lowest, which are an **approximation** of Dart’s operator relationships. You can implement many of these [operators as class members](#).

Description	Operator	Associativity
unary postfix	<code>expr++ expr-- () [] ?[] . ?. !</code>	None
unary prefix	<code>-expr !expr ~expr ++expr --expr await expr</code>	None
multiplicative	<code>* / % ~/</code>	Left
additive	<code>+ -</code>	Left
shift	<code><< >> >>></code>	Left
bitwise AND	<code>&</code>	Left
bitwise XOR	<code>^</code>	Left
bitwise OR	<code> </code>	Left
relational and type test	<code>>= > <= < as is is!</code>	None
equality	<code>== !=</code>	None
logical AND	<code>&&</code>	Left
logical OR	<code> </code>	Left
if null	<code>??</code>	Left
conditional	<code>expr1 ? expr2 : expr3</code>	Right
cascade	<code>.. ?..</code>	Left
assignment	<code>= *= /= += -= &= ^= etc.</code>	Right

 **Warning:** The previous table should only be used as a helpful guide. The notion of operator precedence and associativity is an approximation of the truth found in the language grammar. You can find the authoritative behavior of Dart’s operator relationships in the [Dart language specification](#).

When you use operators, you create expressions. Here are some examples of operator expressions:

```
a++
a + b
a = b
a == b
c ? a : b
a is T
```

In the [operator table](#), each operator has higher precedence than the operators in the rows that follow it. For example, the multiplicative operator `%` has higher precedence than (and thus executes before) the equality operator `==`, which has higher precedence than the logical AND operator `&&`. That precedence means that the following two lines of code execute the same way:

```
// Parentheses improve readability.
if ((n % i == 0) && (d % i == 0)) ...

// Harder to read, but equivalent.
if (n % i == 0 && d % i == 0) ...
```

⚠ Warning: For operators that take two operands, the leftmost operand determines which method is used. For example, if you have a `Vector` object and a `Point` object, then `aVector + aPoint` uses `Vector` addition (+).

Arithmetic operators

Dart supports the usual arithmetic operators, as shown in the following table.

Operator	Meaning
<code>+</code>	Add
<code>-</code>	Subtract
<code>-expr</code>	Unary minus, also known as negation (reverse the sign of the expression)
<code>*</code>	Multiply
<code>/</code>	Divide
<code>~/</code>	Divide, returning an integer result
<code>%</code>	Get the remainder of an integer division (modulo)

Example:

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // Result is a double
assert(5 ~/ 2 == 2); // Result is an int
assert(5 % 2 == 1); // Remainder

assert('5/2 = ${5 ~/ 2} r ${5 % 2}' == '5/2 = 2 r 1');
```

Dart also supports both prefix and postfix increment and decrement operators.

Operator	Meaning
<code>++var</code>	<code>var = var + 1</code> (expression value is <code>var + 1</code>)
<code>var++</code>	<code>var = var + 1</code> (expression value is <code>var</code>)
<code>--var</code>	<code>var = var - 1</code> (expression value is <code>var - 1</code>)
<code>var--</code>	<code>var = var - 1</code> (expression value is <code>var</code>)

Example:

```

int a;
int b;

a = 0;
b = ++a; // Increment a before b gets its value.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Increment a AFTER b gets its value.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decrement a before b gets its value.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decrement a AFTER b gets its value.
assert(a != b); // -1 != 0

```

Equality and relational operators

The following table lists the meanings of equality and relational operators.

Operator	Meaning
<code>==</code>	Equal; see discussion below
<code>!=</code>	Not equal
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

To test whether two objects *x* and *y* represent the same thing, use the `==` operator. (In the rare case where you need to know whether two objects are the exact same object, use the [identical\(\)](#) function instead.) Here's how the `==` operator works:

1. If *x* or *y* is null, return true if both are null, and false if only one is null.
2. Return the result of invoking the `==` method on *x* with the argument *y*. (That's right, operators such as `==` are methods that are invoked on their first operand. For details, see [Operators](#).)

Here's an example of using each of the equality and relational operators:

```

assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);

```

Type test operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Operator	Meaning
<code>as</code>	Typecast (also used to specify library prefixes)

Operator	Meaning
<code>is</code>	True if the object has the specified type
<code>is!</code>	True if the object doesn't have the specified type

The result of `obj is T` is true if `obj` implements the interface specified by `T`. For example, `obj is Object?` is always true.

Use the `as` operator to cast an object to a particular type if and only if you are sure that the object is of that type. Example:

```
(employee as Person).firstName = 'Bob';
```

If you aren't sure that the object is of type `T`, then use `is T` to check the type before using the object.

```
if (employee is Person) {
    // Type check
    employee.firstName = 'Bob';
}
```

Note: The code isn't equivalent. If `employee` is null or not a `Person`, the first example throws an exception; the second does nothing.

Assignment operators

As you've already seen, you can assign values using the `=` operator. To assign only if the assigned-to variable is null, use the `??=` operator.

```
// Assign value to a
a = value;
// Assign value to b if b is null; otherwise, b stays the same
b ??= value;
```

Compound assignment operators such as `+=` combine an operation with an assignment.

<code>=</code>	<code>*=</code>	<code>%=</code>	<code>>>>=</code>	<code>^=</code>
<code>+=</code>	<code>/=</code>	<code><<=</code>	<code>&=</code>	<code> =</code>
<code>-=</code>	<code>~/=</code>	<code>>>=</code>		

Here's how compound assignment operators work:

	Compound assignment	Equivalent expression
For an operator <i>op</i> :	<code>a op= b</code>	<code>a = a op b</code>
Example:	<code>a += b</code>	<code>a = a + b</code>

The following example uses assignment and compound assignment operators:

```
var a = 2; // Assign using =
a *= 3; // Assign and multiply: a = a * 3
assert(a == 6);
```

Logical operators

You can invert or combine boolean expressions using the logical operators.

Operator	Meaning
<code>!expr</code>	inverts the following expression (changes false to true, and vice versa)
<code> </code>	logical OR
<code>&&</code>	logical AND

Here's an example of using the logical operators:

```
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

Bitwise and shift operators


You can manipulate the individual bits of numbers in Dart. Usually, you'd use these bitwise and shift operators with integers.

Operator	Meaning
<code>&</code>	AND
<code> </code>	OR
<code>^</code>	XOR
<code>~expr</code>	Unary bitwise complement (0s become 1s; 1s become 0s)
<code><<</code>	Shift left
<code>>></code>	Shift right
<code>>>></code>	Unsigned shift right

Here's an example of using bitwise and shift operators:

```
final value = 0x22;
final bitmask = 0x0f;

assert((value & bitmask) == 0x02); // AND
assert((value & ~bitmask) == 0x20); // AND NOT
assert((value | bitmask) == 0x2f); // OR
assert((value ^ bitmask) == 0x2d); // XOR
assert((value << 4) == 0x220); // Shift left
assert((value >> 4) == 0x02); // Shift right
assert((value >>> 4) == 0x02); // Unsigned shift right
assert((-value >> 4) == -0x03); // Shift right
assert((-value >>> 4) > 0); // Unsigned shift right
```


 **Version note:** The `>>>` operator (known as *triple-shift* or *unsigned shift*) requires a [language version](#) of at least 2.14.

Conditional expressions

Dart has two operators that let you concisely evaluate expressions that might otherwise require [if-else](#) statements:

`condition ? expr1 : expr2`

If `condition` is true, evaluates `expr1` (and returns its value); otherwise, evaluates and returns the value of `expr2`.

`expr1 ?? expr2`

If `expr1` is non-null, returns its value; otherwise, evaluates and returns the value of `expr2`.

When you need to assign a value based on a boolean expression, consider using `?` and `:`.

```
var visibility = isPublic ? 'public' : 'private';
```

If the boolean expression tests for null, consider using `??`.

```
String playerName(String? name) => name ?? 'Guest';
```

The previous example could have been written at least two other ways, but not as succinctly:

```
// Slightly longer version uses ?: operator.
String playerName(String? name) => name != null ? name : 'Guest';

// Very long version uses if-else statement.
String playerName(String? name) {
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
}
```

Cascade notation

Cascades (`..`, `?..`) allow you to make a sequence of operations on the same object. In addition to accessing instance members, you can also call instance methods on that same object. This often saves you the step of creating a temporary variable and allows you to write more fluid code.

Consider the following code:

```
var paint = Paint()
  ..color = Colors.black
  ..strokeCap = StrokeCap.round
  ..strokeWidth = 5.0;
```

The constructor, `Paint()`, returns a `Paint` object. The code that follows the cascade notation operates on this object, ignoring any values that might be returned.

The previous example is equivalent to this code:

```
var paint = Paint();
paint.color = Colors.black;
paint.strokeCap = StrokeCap.round;
paint.strokeWidth = 5.0;
```

If the object that the cascade operates on can be null, then use a *null-shorting* cascade (`?..`) for the first operation. Starting with `?..` guarantees that none of the cascade operations are attempted on that null object.

```
querySelector('#confirm') // Get an object.
  ?.text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'))
  ..scrollIntoView();
```

 **Version note:** The `?..` syntax requires a [language version](#) of at least 2.12.

The previous code is equivalent to the following:

```
var button = querySelector('#confirm');
button?.text = 'Confirm';
button?.classes.add('important');
button?.onClick.listen((e) => window.alert('Confirmed!'));
button?.scrollIntoView();
```


You can also nest cascades. For example:

```
final addressBook = (AddressBookBuilder()
  ..name = 'jenny'
  ..email = 'jenny@example.com'
  ..phone = (PhoneNumberBuilder()
    ..number = '415-555-0100'
    ..label = 'home')
  .build())
  .build();
```

Be careful to construct your cascade on a function that returns an actual object. For example, the following code fails:

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: method 'write' isn't defined for 'void'.
```

The `sb.write()` call returns void, and you can't construct a cascade on `void`.

 **Note:** Strictly speaking, the “double dot” notation for cascades isn't an operator. It's just part of the Dart syntax.

Other operators

You've seen most of the remaining operators in other examples:

Operator	Name	Meaning
()	Function application	Represents a function call
[]	Subscript access	Represents a call to the overridable [] operator; example: <code>fooList[1]</code> passes the int <code>1</code> to <code>fooList</code> to access the element at index <code>1</code>
?[]	Conditional subscript access	Like [], but the leftmost operand can be null; example: <code>fooList?[1]</code> passes the int <code>1</code> to <code>fooList</code> to access the element at index <code>1</code> unless <code>fooList</code> is null (in which case the expression evaluates to null)
.	Member access	Refers to a property of an expression; example: <code>foo.bar</code> selects property <code>bar</code> from expression <code>foo</code>
?.	Conditional member access	Like ., but the leftmost operand can be null; example: <code>foo?.bar</code> selects property <code>bar</code> from expression <code>foo</code> unless <code>foo</code> is null (in which case the value of <code>foo?.bar</code> is null)
!	Null assertion operator	Casts an expression to its underlying non-nullable type, throwing a runtime exception if the cast fails; example: <code>foo!.bar</code> asserts <code>foo</code> is non-null and selects the property <code>bar</code> , unless <code>foo</code> is null in which case a runtime exception is thrown

For more information about the `.`, `?.`, and `!.` operators, see [Classes](#).

Control flow statements

You can control the flow of your Dart code using any of the following:

- `if` and `else`
- `for` loops
- `while` and `do-while` loops
- `break` and `continue`
- `switch` and `case`
- `assert`

You can also affect the control flow using `try-catch` and `throw`, as explained in [Exceptions](#).

If and else

Dart supports `if` statements with optional `else` statements, as the next sample shows. Also see [conditional expressions](#).

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

The statement conditions must be expressions that evaluate to boolean values, nothing else. See [Booleans](#) for more information.

For loops

You can iterate with the standard `for` loop. For example:

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

Closures inside of Dart's `for` loops capture the *value* of the index, avoiding a common pitfall found in JavaScript. For example, consider:

```
var callbacks = [];
for (var i = 0; i < 2; i++) {
  callbacks.add(() => print(i));
}

for (final c in callbacks) {
  c();
}
```

The output is `0` and then `1`, as expected. In contrast, the example would print `2` and then `2` in JavaScript.

If the object that you are iterating over is an Iterable (such as List or Set) and if you don't need to know the current iteration counter, you can use the `for-in` form of [iteration](#):

```
for (final candidate in candidates) {
  candidate.interview();
}
```

💡 **Tip:** To practice using `for-in`, follow the [Iterable collections codelab](#).

Iterable classes also have a [forEach\(\)](#) method as another option:

```
var collection = [1, 2, 3];
collection.forEach(print); // 1 2 3
```

While and do-while

A `while` loop evaluates the condition before the loop:

```
while (!isDone()) {
  doSomething();
}
```

A `do-while` loop evaluates the condition *after* the loop:

```
do {
  printLine();
} while (!atEndOfPage());
```

Break and continue

Use `break` to stop looping:

```
while (true) {  
  if (shutdownRequested()) break;  
  processIncomingRequests();  
}
```

Use `continue` to skip to the next loop iteration:

```
for (int i = 0; i < candidates.length; i++) {  
  var candidate = candidates[i];  
  if (candidate.yearsExperience < 5) {  
    continue;  
  }  
  candidate.interview();  
}
```

You might write that example differently if you're using an [Iterable](#) such as a list or set:

```
candidates  
  .where((c) => c.yearsExperience >= 5)  
  .forEach((c) => c.interview());
```

Switch and case

Switch statements in Dart compare integer, string, or compile-time constants using `==`. The compared objects must all be instances of the same class (and not of any of its subtypes), and the class must not override `==`. [Enumerated types](#) work well in `switch` statements.

Each non-empty `case` clause ends with a `break` statement, as a rule. Other valid ways to end a non-empty `case` clause are a `continue`, `throw`, or `return` statement.

Use a `default` clause to execute code when no `case` clause matches:

```
var command = 'OPEN';  
switch (command) {  
  case 'CLOSED':  
    executeClosed();  
    break;  
  case 'PENDING':  
    executePending();  
    break;  
  case 'APPROVED':  
    executeApproved();  
    break;  
  case 'DENIED':  
    executeDenied();  
    break;  
  case 'OPEN':  
    executeOpen();  
    break;  
  default:  
    executeUnknown();  
}
```

The following example omits the `break` statement in a `case` clause, thus generating an error:

```

var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break

  case 'CLOSED':
    executeClosed();
    break;
}

```

However, Dart does support empty `case` clauses, allowing a form of fall-through:

```

var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // Empty case falls through.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}

```

If you really want fall-through, you can use a `continue` statement and a label:

```

var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed;
    // Continues executing at the nowClosed label.

  nowClosed:
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}

```

A `case` clause can have local variables, which are visible only inside the scope of that clause.

Assert

During development, use an assert statement—`assert(condition, optionalMessage);`—to disrupt normal execution if a boolean condition is false. You can find examples of assert statements throughout this tour. Here are some more:

```

// Make sure the variable has a non-null value.
assert(text != null);

// Make sure the value is less than 100.
assert(number < 100);

// Make sure this is an https URL.
assert(urlString.startsWith('https'));

```

To attach a message to an assertion, add a string as the second argument to `assert` (optionally with a [trailing comma](#)):

```

assert(urlString.startsWith('https'),
  'URL ($urlString) should start with "https".');

```

The first argument to `assert` can be any expression that resolves to a boolean value. If the expression's value is true, the assertion succeeds and execution continues. If it's false, the assertion fails and an exception (an [AssertionError](#)) is thrown.

When exactly do assertions work? That depends on the tools and framework you're using:

- Flutter enables assertions in [debug mode](#).
- Development-only tools such as `[webdev serve]` typically enable assertions by default.
- Some tools, such as `[dart run]` and `[dart compile js]` support assertions through a command-line flag: `--enable-asserts`.

In production code, assertions are ignored, and the arguments to `assert` aren't evaluated.

Exceptions

Your Dart code can throw and catch exceptions. Exceptions are errors indicating that something unexpected happened. If the exception isn't caught, the [isolate](#) that raised the exception is suspended, and typically the isolate and its program are terminated.

In contrast to Java, all of Dart's exceptions are unchecked exceptions. Methods don't declare which exceptions they might throw, and you aren't required to catch any exceptions.

Dart provides [Exception](#) and [Error](#) types, as well as numerous predefined subtypes. You can, of course, define your own exceptions. However, Dart programs can throw any non-null object—not just `Exception` and `Error` objects—as an exception.

Throw

Here's an example of throwing, or *raising*, an exception:

```
throw FormatException('Expected at least 1 section');
```

You can also throw arbitrary objects:

```
throw 'Out of llamas!';
```

Note: Production-quality code usually throws types that implement [Error](#) or [Exception](#).

Because throwing an exception is an expression, you can throw exceptions in `=>` statements, as well as anywhere else that allows expressions:

```
void distanceTo(Point other) => throw UnimplementedError();
```

Catch

Catching, or capturing, an exception stops the exception from propagating (unless you rethrow the exception). Catching an exception gives you a chance to handle it:

```
try {  
  breedMoreLlamas();  
} on OutOfLlamasException {  
  buyMoreLlamas();  
}
```

To handle code that can throw more than one type of exception, you can specify multiple catch clauses. The first catch clause that matches the thrown object's type handles the exception. If the catch clause does not specify a type, that clause can handle any type of thrown object:

```
try {
    breedMoreLlamas();
} on OutOfLlamasException {
    // A specific exception
    buyMoreLlamas();
} on Exception catch (e) {
    // Anything else that is an exception
    print('Unknown exception: $e');
} catch (e) {
    // No specified type, handles all
    print('Something really unknown: $e');
}
```

As the preceding code shows, you can use either `on` or `catch` or both. Use `on` when you need to specify the exception type. Use `catch` when your exception handler needs the exception object.

You can specify one or two parameters to `catch()`. The first is the exception that was thrown, and the second is the stack trace (a [StackTrace](#) object).

```
try {
    // ...
} on Exception catch (e) {
    print('Exception details:\n $e');
} catch (e, s) {
    print('Exception details:\n $e');
    print('Stack trace:\n $s');
}
```

To partially handle an exception, while allowing it to propagate, use the `rethrow` keyword.

```
void misbehave() {
    try {
        dynamic foo = true;
        print(foo++); // Runtime error
    } catch (e) {
        print('misbehave() partially handled ${e.runtimeType}.');
        rethrow; // Allow callers to see the exception.
    }
}

void main() {
    try {
        misbehave();
    } catch (e) {
        print('main() finished handling ${e.runtimeType}.');
    }
}
```

Finally

To ensure that some code runs whether or not an exception is thrown, use a `finally` clause. If no `catch` clause matches the exception, the exception is propagated after the `finally` clause runs:


```

try {
    breedMoreLlamas();
} finally {
    // Always clean up, even if an exception is thrown.
    cleanLlamaStalls();
}

```

The `finally` clause runs after any matching `catch` clauses:

```

try {
    breedMoreLlamas();
} catch (e) {
    print('Error: $e'); // Handle the exception first.
} finally {
    cleanLlamaStalls(); // Then clean up.
}

```

Learn more by reading the [Exceptions](#) section of the library tour.

Classes

Dart is an object-oriented language with classes and mixin-based inheritance. Every object is an instance of a class, and all classes except `Null` descend from `Object`. *Mixin-based inheritance* means that although every class (except for the [top class](#), `Object?`) has exactly one superclass, a class body can be reused in multiple class hierarchies. [Extension methods](#) are a way to add functionality to a class without changing the class or creating a subclass.

Using class members

Objects have *members* consisting of functions and data (*methods* and *instance variables*, respectively). When you call a method, you *invoke* it on an object: the method has access to that object's functions and data.

Use a dot (`.`) to refer to an instance variable or method:

```

var p = Point(2, 2);

// Get the value of y.
assert(p.y == 2);

// Invoke distanceTo() on p.
double distance = p.distanceTo(Point(4, 4));

```

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null:

```

// If p is non-null, set a variable equal to its y value.
var a = p?.y;

```

Using constructors

You can create an object using a *constructor*. Constructor names can be either *ClassName* or *ClassName.identifier*. For example, the following code creates `Point` objects using the `Point()` and `Point.fromJson()` constructors:

```

var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});

```

The following code has the same effect, but uses the optional `new` keyword before the constructor name:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

Some classes provide [constant constructors](#). To create a compile-time constant using a constant constructor, put the `const` keyword before the constructor name:

```
var p = const ImmutablePoint(2, 2);
```

Constructing two identical compile-time constants results in a single, canonical instance:

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a, b)); // They are the same instance!
```

Within a *constant context*, you can omit the `const` before a constructor or literal. For example, look at this code, which creates a const map:

```
// Lots of const keywords here.
const pointAndLine = const {
  'point': const [const ImmutablePoint(0, 0)],
  'line': const [const ImmutablePoint(1, 10), const ImmutablePoint(-2, 11)],
};
```

You can omit all but the first use of the `const` keyword:

```
// Only one const, which establishes the constant context.
const pointAndLine = {
  'point': [ImmutablePoint(0, 0)],
  'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};
```

If a constant constructor is outside of a constant context and is invoked without `const`, it creates a **non-constant object**:

```
var a = const ImmutablePoint(1, 1); // Creates a constant
var b = ImmutablePoint(1, 1); // Does NOT create a constant

assert(!identical(a, b)); // NOT the same instance!
```

Getting an object's type

To get an object's type at runtime, you can use the `Object` property `runtimeType`, which returns a [Type](#) object.

```
print('The type of a is ${a.runtimeType}');
```

⚠ Use a [type test operator](#) rather than `runtimeType` to test an object's type. In production environments, the test `object is Type` is more stable than the test `object.runtimeType == Type`.

Up to here, you've seen how to *use* classes. The rest of this section shows how to *implement* classes.

Instance variables

Here's how you declare instance variables:

```
class Point {  
    double? x; // Declare instance variable x, initially null.  
    double? y; // Declare y, initially null.  
    double z = 0; // Declare z, initially 0.  
}
```

All uninitialized instance variables have the value `null`.

All instance variables generate an implicit *getter* method. Non-final instance variables and `late final` instance variables without initializers also generate an implicit *setter* method. For details, see [Getters and setters](#).

If you initialize a non-`late` instance variable where it's declared, the value is set when the instance is created, which is before the constructor and its initializer list execute. As a result, non-`late` instance variable initializers can't access `this`.

```
class Point {  
    double? x; // Declare instance variable x, initially null.  
    double? y; // Declare y, initially null.  
}  
  
void main() {  
    var point = Point();  
    point.x = 4; // Use the setter method for x.  
    assert(point.x == 4); // Use the getter method for x.  
    assert(point.y == null); // Values default to null.  
}
```

Instance variables can be `final`, in which case they must be set exactly once. Initialize `final`, non-`late` instance variables at declaration, using a constructor parameter, or using a constructor's [initializer list](#):

```
class ProfileMark {  
    final String name;  
    final DateTime start = DateTime.now();  
  
    ProfileMark(this.name);  
    ProfileMark.unnamed() : name = '';  
}
```

If you need to assign the value of a `final` instance variable after the constructor body starts, you can use one of the following:

- Use a [factory constructor](#).
- Use `late final`, but [be careful](#): a `late final` without an initializer adds a setter to the API.

Constructors

Declare a constructor by creating a function with the same name as its class (plus, optionally, an additional identifier as described in [Named constructors](#)). The most common form of constructor, the generative constructor, creates a new instance of a class:

```

class Point {
  double x = 0;
  double y = 0;

  Point(double x, double y) {
    // See initializing formal parameters for a better way
    // to initialize instance variables.
    this.x = x;
    this.y = y;
  }
}

```

The `this` keyword refers to the current instance.

Note: Use `this` only when there is a name conflict. Otherwise, Dart style omits the `this`.

Initializing formal parameters

The pattern of assigning a constructor argument to an instance variable is so common, Dart has initializing formal parameters to make it easy.

Initializing parameters can also be used to initialize non-nullable or `final` instance variables, which both must be initialized or provided a default value.

```

class Point {
  final double x;
  final double y;

  // Sets the x and y instance variables
  // before the constructor body runs.
  Point(this.x, this.y);
}

```

The variables introduced by the initializing formals are implicitly final and only in scope of the initializer list.

Default constructors

If you don't declare a constructor, a default constructor is provided for you. The default constructor has no arguments and invokes the no-argument constructor in the superclass.

Constructors aren't inherited

Subclasses don't inherit constructors from their superclass. A subclass that declares no constructors has only the default (no argument, no name) constructor.

Named constructors

Use a named constructor to implement multiple constructors for a class or to provide extra clarity:

```

const double xOrigin = 0;
const double yOrigin = 0;

class Point {
    final double x;
    final double y;

    Point(this.x, this.y);

    // Named constructor
    Point.origin()
        : x = xOrigin,
          y = yOrigin;
}

```

Remember that constructors are not inherited, which means that a superclass's named constructor is not inherited by a subclass. If you want a subclass to be created with a named constructor defined in the superclass, you must implement that constructor in the subclass.

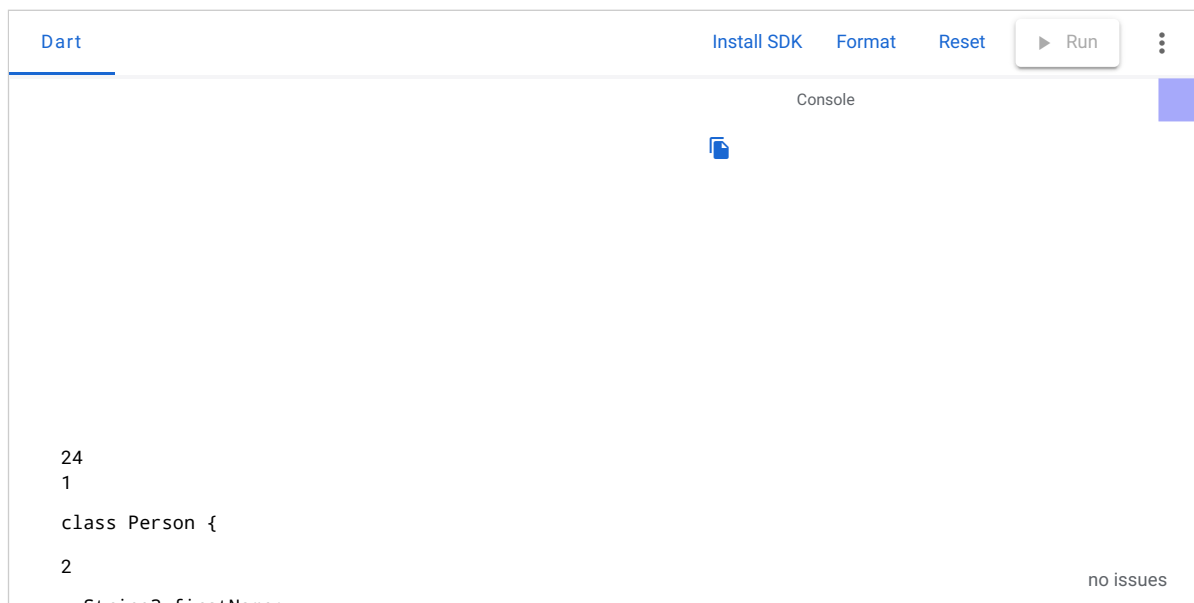
Invoking a non-default superclass constructor

By default, a constructor in a subclass calls the superclass's unnamed, no-argument constructor. The superclass's constructor is called at the beginning of the constructor body. If an [initializer list](#) is also being used, it executes before the superclass is called. In summary, the order of execution is as follows:

1. initializer list
2. superclass's no-arg constructor
3. main class's no-arg constructor

If the superclass doesn't have an unnamed, no-argument constructor, then you must manually call one of the constructors in the superclass. Specify the superclass constructor after a colon (:), just before the constructor body (if any).

In the following example, the constructor for the `Employee` class calls the named constructor for its superclass, `Person`. Click **Run** to execute the code.



Because the arguments to the superclass constructor are evaluated before invoking the constructor, an argument can be an expression such as a function call:

```

class Employee extends Person {
    Employee() : super.fromJson(fetchDefaultData());
    // ...
}

```

⚠ Warning: Arguments to the superclass constructor don't have access to `this`. For example, arguments can call static methods but not instance methods.

To avoid having to manually pass each parameter into the super invocation of a constructor, you can use super-initializer parameters to forward parameters to the specified or default superclass constructor. This feature can't be used with redirecting constructors. Super-initializer parameters have similar syntax and semantics to [initializing formal parameters](#):

```
class Vector2d {
    final double x;
    final double y;

    Vector2d(this.x, this.y);
}

class Vector3d extends Vector2d {
    final double z;

    // Forward the x and y parameters to the default super constructor like:
    // Vector3d(final double x, final double y, this.z) : super(x, y);
    Vector3d(super.x, super.y, this.z);
}
```

Super-initializer parameters cannot be positional if the super-constructor invocation already has positional arguments, but they can always be named:

```
class Vector2d {
    // ...

    Vector2d.named({required this.x, required this.y});
}

class Vector3d extends Vector2d {
    // ...

    // Forward the y parameter to the named super constructor like:
    // Vector3d.yzPlane({required double y, required this.z})
    //       : super.named(x: 0, y: y);
    Vector3d.yzPlane({required super.y, required this.z}) : super.named(x: 0);
}
```

🔗 Version note: Using super-initializer parameters requires a [language version](#) of at least 2.17. If you're using an earlier language version, you must manually pass in all super constructor parameters.

Initializer list

Besides invoking a superclass constructor, you can also initialize instance variables before the constructor body runs. Separate initializers with commas.

```
// Initializer list sets instance variables before
// the constructor body runs.
Point.fromJson(Map<String, double> json)
    : x = json['x']!,
      y = json['y']! {
    print('In Point.fromJson(): ($x, $y)');
}
```

Warning: The right-hand side of an initializer doesn't have access to `this`.

During development, you can validate inputs by using `assert` in the initializer list.

```
Point.withAssert(this.x, this.y) : assert(x >= 0) {  
  print('In Point.withAssert(): ($x, $y)');  
}
```

Initializer lists are handy when setting up final fields. The following example initializes three final fields in an initializer list. Click **Run** to execute the code.

The screenshot shows an IDE window titled "Dart" with a toolbar containing "Install SDK", "Format", "Reset", and a "Run" button. Below the toolbar is a "Console" window. The code editor shows the following Dart code:

```
17  
1  
import 'dart:math';
```

The console window displays two error messages:

- line 9 • Use an initializing formal to assign a parameter to a field. ([view docs](#))
Try using an initialing formal ('this.x') to initialize the field.
- line 10 • Use an initializing formal to assign a parameter to a field. ([view docs](#))
Try using an initialing formal ('this.y') to initialize the field.

At the bottom right of the console, it says "2 issues" and a "hide" button.

Redirecting constructors

Sometimes a constructor's only purpose is to redirect to another constructor in the same class. A redirecting constructor's body is empty, with the constructor call (using `this` instead of the class name) appearing after a colon (:).

```
class Point {  
  double x, y;  
  
  // The main constructor for this class.  
  Point(this.x, this.y);  
  
  // Delegates to the main constructor.  
  Point.alongXAxis(double x) : this(x, 0);  
}
```

Constant constructors

If your class produces objects that never change, you can make these objects compile-time constants. To do this, define a `const` constructor and make sure that all instance variables are `final`.

```
class ImmutablePoint {  
  static const ImmutablePoint origin = ImmutablePoint(0, 0);  
  
  final double x, y;  
  
  const ImmutablePoint(this.x, this.y);  
}
```

Constant constructors don't always create constants. For details, see the section on [using constructors](#).

Factory constructors

Use the `factory` keyword when implementing a constructor that doesn't always create a new instance of its class. For example, a factory constructor might return an instance from a cache, or it might return an instance of a subtype. Another use case for factory constructors is initializing a final variable using logic that can't be handled in the initializer list.

💡 **Tip:** Another way to handle late initialization of a final variable is to [use late final \(carefully!\)](#).

In the following example, the `Logger` factory constructor returns objects from a cache, and the `Logger.fromJson` factory constructor initializes a final variable from a JSON object.

```
class Logger {
    final String name;
    bool mute = false;

    // _cache is library-private, thanks to
    // the _ in front of its name.
    static final Map<String, Logger> _cache = <String, Logger>{};

    factory Logger(String name) {
        return _cache.putIfAbsent(name, () => Logger._internal(name));
    }

    factory Logger.fromJson(Map<String, Object> json) {
        return Logger(json['name'].toString());
    }

    Logger._internal(this.name);

    void log(String msg) {
        if (!mute) print(msg);
    }
}
```

❗ **Note:** Factory constructors have no access to `this`.

Invoke a factory constructor just like you would any other constructor:

```
var logger = Logger('UI');
logger.log('Button clicked');

var logMap = {'name': 'UI'};
var loggerJson = Logger.fromJson(logMap);
```

Methods

Methods are functions that provide behavior for an object.

Instance methods

Instance methods on objects can access instance variables and `this`. The `distanceTo()` method in the following sample is an example of an instance method:


```
import 'dart:math';

class Point {
  final double x;
  final double y;

  Point(this.x, this.y);

  double distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

Operators

Operators are instance methods with special names. Dart allows you to define operators with the following names:

<	+		>>>
>	/	^	[]
<=	~/	&	[]=
>=	*	<<	~
-	%	>>	==

Note: You may have noticed that some [operators](#), like `!=`, aren't in the list of names. That's because they're just syntactic sugar. For example, the expression `e1 != e2` is syntactic sugar for `!(e1 == e2)`.

An operator declaration is identified using the built-in identifier `operator`. The following example defines vector addition (+), subtraction (-), and equality (==):

```
class Vector {
  final int x, y;

  Vector(this.x, this.y);

  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

  @override
  bool operator ==(Object other) =>
    other is Vector && x == other.x && y == other.y;

  @override
  int get hashCode => Object.hash(x, y);
}

void main() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);

  assert(v + w == Vector(4, 5));
  assert(v - w == Vector(0, 1));
}
```

Getters and setters

Getters and setters are special methods that provide read and write access to an object's properties. Recall that each instance variable has an implicit getter, plus a setter if appropriate. You can create additional properties by implementing getters and setters, using the `get` and `set` keywords:

```
class Rectangle {
    double left, top, width, height;

    Rectangle(this.left, this.top, this.width, this.height);

    // Define two calculated properties: right and bottom.
    double get right => left + width;
    set right(double value) => left = value - width;
    double get bottom => top + height;
    set bottom(double value) => top = value - height;
}

void main() {
    var rect = Rectangle(3, 4, 20, 15);
    assert(rect.left == 3);
    rect.right = 12;
    assert(rect.left == -8);
}
```

With getters and setters, you can start with instance variables, later wrapping them with methods, all without changing client code.

Note: Operators such as increment (`++`) work in the expected way, whether or not a getter is explicitly defined. To avoid any unexpected side effects, the operator calls the getter exactly once, saving its value in a temporary variable.

Abstract methods

Instance, getter, and setter methods can be abstract, defining an interface but leaving its implementation up to other classes. Abstract methods can only exist in [abstract classes](#).

To make a method abstract, use a semicolon (;) instead of a method body:

```
abstract class Doer {
    // Define instance variables and methods...

    void doSomething(); // Define an abstract method.
}

class EffectiveDoer extends Doer {
    void doSomething() {
        // Provide an implementation, so the method is not abstract here...
    }
}
```

Abstract classes

Use the `abstract` modifier to define an *abstract class*—a class that can't be instantiated. Abstract classes are useful for defining interfaces, often with some implementation. If you want your abstract class to appear to be instantiable, define a [factory constructor](#).

Abstract classes often have [abstract methods](#). Here's an example of declaring an abstract class that has an abstract method:

```
// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
    // Define constructors, fields, methods...

    void updateChildren(); // Abstract method.
}
```

Implicit interfaces

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface.

A class implements one or more interfaces by declaring them in an `implements` clause and then providing the APIs required by the interfaces. For example:

```
// A person. The implicit interface contains greet().
class Person {
    // In the interface, but visible only in this library.
    final String _name;

    // Not in the interface, since this is a constructor.
    Person(this._name);

    // In the interface.
    String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface.
class Impostor implements Person {
    String get _name => '';

    String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
    print(greetBob(Person('Kathy')));
    print(greetBob(Impostor()));
}
```

Here's an example of specifying that a class implements multiple interfaces:

```
class Point implements Comparable, Location {...}
```

Extending a class

Use `extends` to create a subclass, and `super` to refer to the superclass:

```

class Television {
  void turnOn() {
    _illuminateDisplay();
    _activateIrSensor();
  }
  // ...
}

class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    _bootNetworkInterface();
    _initializeMemory();
    _upgradeApps();
  }
  // ...
}

```

For another usage of `extends`, see the discussion of [parameterized types](#) in [generics](#).

Overriding members

Subclasses can override instance methods (including [operators](#)), getters, and setters. You can use the `@override` annotation to indicate that you are intentionally overriding a member:

```

class Television {
  // ...
  set contrast(int value) {...}
}

class SmartTelevision extends Television {
  @override
  set contrast(num value) {...}
  // ...
}

```

An overriding method declaration must match the method (or methods) that it overrides in several ways:

- The return type must be the same type as (or a subtype of) the overridden method's return type.
- Argument types must be the same type as (or a supertype of) the overridden method's argument types. In the preceding example, the `contrast` setter of `SmartTelevision` changes the argument type from `int` to a supertype, `num`.
- If the overridden method accepts n positional parameters, then the overriding method must also accept n positional parameters.
- A [generic method](#) can't override a non-generic one, and a non-generic method can't override a generic one.

Sometimes you might want to narrow the type of a method parameter or an instance variable. This violates the normal rules, and it's similar to a downcast in that it can cause a type error at runtime. Still, narrowing the type is possible if the code can guarantee that a type error won't occur. In this case, you can use the [covariant keyword](#) in a parameter declaration. For details, see the [Dart language specification](#).

⚠ Warning: If you override `==`, you should also override `Object`'s `hashCode` getter. For an example of overriding `==` and `hashCode`, see [Implementing map keys](#).

noSuchMethod()

To detect or react whenever code attempts to use a non-existent method or instance variable, you can override `noSuchMethod()`:

```
class A {
  // Unless you override noSuchMethod, using a
  // non-existent member results in a NoSuchMethodError.
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: '
      '${invocation.memberName}');
  }
}
```

You **can't invoke** an unimplemented method unless **one** of the following is true:

- The receiver has the static type `dynamic`.
- The receiver has a static type that defines the unimplemented method (abstract is OK), and the dynamic type of the receiver has an implementation of `noSuchMethod()` that's different from the one in class `Object`.

For more information, see the informal [noSuchMethod forwarding specification](#).

Extension methods

Extension methods are a way to add functionality to existing libraries. You might use extension methods without even knowing it. For example, when you use code completion in an IDE, it suggests extension methods alongside regular methods.

Here's an example of using an extension method on `String` named `parseInt()` that's defined in `string_api.dart`:

```
import 'string_api.dart';
...
print('42'.padLeft(5)); // Use a String method.
print('42'.parseInt()); // Use an extension method.
```

For details of using and implementing extension methods, see the [extension methods page](#).

Enumerated types

Enumerated types, often called *enumerations* or *enums*, are a special kind of class used to represent a fixed number of constant values.

Note: All enums automatically extend the `Enum` class. They are also sealed, meaning they cannot be subclassed, implemented, mixed in, or otherwise explicitly instantiated.

Abstract classes and mixins can explicitly implement or extend `Enum`, but unless they are then implemented by or mixed into an enum declaration, no objects can actually implement the type of that class or mixin.

Declaring simple enums

To declare a simple enumerated type, use the `enum` keyword and list the values you want to be enumerated:

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

Tip: You can also use [trailing commas](#) when declaring an enumerated type to help prevent copy-paste errors.

Declaring enhanced enums

Dart also allows enum declarations to declare classes with fields, methods, and const constructors which are limited to a fixed number of known constant instances.


To declare an enhanced enum, follow a syntax similar to normal [classes](#), but with a few extra requirements:

- Instance variables must be `final`, including those added by [mixins](#).
- All [generative constructors](#) must be constant.
- [Factory constructors](#) can only return one of the fixed, known enum instances.
- No other class can be extended as `Enum` is automatically extended.
- There cannot be overrides for `index`, `hashCode`, the equality operator `==`.
- A member named `values` cannot be declared in an enum, as it would conflict with the automatically generated static `values` getter.
- All instances of the enum must be declared in the beginning of the declaration, and there must be at least one instance declared.

Here is an example that declares an enhanced enum with multiple instances, instance variables, a getter, and an implemented interface:

```
enum Vehicle implements Comparable<Vehicle> {  
    car(tires: 4, passengers: 5, carbonPerKilometer: 400),  
    bus(tires: 6, passengers: 50, carbonPerKilometer: 800),  
    bicycle(tires: 2, passengers: 1, carbonPerKilometer: 0);  
  
    const Vehicle({  
        required this.tires,  
        required this.passengers,  
        required this.carbonPerKilometer,  
    });  
  
    final int tires;  
    final int passengers;  
    final int carbonPerKilometer;  
  
    int get carbonFootprint => (carbonPerKilometer / passengers).round();  
  
    @override  
    int compareTo(Vehicle other) => carbonFootprint - other.carbonFootprint;  
}
```

To learn more about declaring enhanced enums, see the section on [Classes](#).

 **Version note:** Enhanced enums require a [language version](#) of at least 2.17.

Using enums

Access the enumerated values like any other [static variable](#):

```
final favoriteColor = Color.blue;  
if (favoriteColor == Color.blue) {  
    print('Your favorite color is blue!');  
}
```

Each value in an enum has an `index` getter, which returns the zero-based position of the value in the enum declaration. For example, the first value has index 0, and the second value has index 1.

```
assert(Color.red.index == 0);  
assert(Color.green.index == 1);  
assert(Color.blue.index == 2);
```

To get a list of all the enumerated values, use the enum's `values` constant.

```
List<Color> colors = Color.values;
assert(colors[2] == Color.blue);
```

You can use enums in [switch statements](#), and you'll get a warning if you don't handle all of the enum's values:

```
var aColor = Color.blue;

switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
    print('Green as grass!');
    break;
  default: // Without this, you see a WARNING.
    print(aColor); // 'Color.blue'
}
```

If you need to access the name of an enumerated value, such as 'blue' from `Color.blue`, use the `.name` property:

```
print(Color.blue.name); // 'blue'
```

Adding features to a class: mixins

Mixins are a way of reusing a class's code in multiple class hierarchies.

To *use* a mixin, use the `with` keyword followed by one or more mixin names. The following example shows two classes that use mixins:

```
class Musician extends Performer with Musical {
  // ...
}

class Maestro extends Person with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

To *implement* a mixin, create a class that extends `Object` and declares no constructors. Unless you want your mixin to be usable as a regular class, use the `mixin` keyword instead of `class`. For example:

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

Sometimes you might want to restrict the types that can use a mixin. For example, the mixin might depend on being able to invoke a method that the mixin doesn't define. As the following example shows, you can restrict a mixin's use by using the `on` keyword to specify the required superclass:

```
class Musician {
    // ...
}
mixin MusicalPerformer on Musician {
    // ...
}
class SingerDancer extends Musician with MusicalPerformer {
    // ...
}
```

In the preceding code, only classes that extend or implement the `Musician` class can use the mixin `MusicalPerformer`. Because `SingerDancer` extends `Musician`, `SingerDancer` can mix in `MusicalPerformer`.

Class variables and methods

Use the `static` keyword to implement class-wide variables and methods.

Static variables

Static variables (class variables) are useful for class-wide state and constants:

```
class Queue {
    static const initialCapacity = 16;
    // ...
}

void main() {
    assert(Queue.initialCapacity == 16);
}
```

Static variables aren't initialized until they're used.

Note: This page follows the [style guide recommendation](#) of preferring `lowerCamelCase` for constant names.

Static methods

Static methods (class methods) don't operate on an instance, and thus don't have access to `this`. They do, however, have access to static variables. As the following example shows, you invoke static methods directly on a class:


```
import 'dart:math';

class Point {
  double x, y;
  Point(this.x, this.y);

  static double distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

Note: Consider using top-level functions, instead of static methods, for common or widely used utilities and functionality.

You can use static methods as compile-time constants. For example, you can pass a static method as a parameter to a constant constructor.

Generics

If you look at the API documentation for the basic array type, [List](#), you'll see that the type is actually `List<E>`. The `<...>` notation marks List as a *generic* (or *parameterized*) type—a type that has formal type parameters. [By convention](#), most type variables have single-letter names, such as E, T, S, K, and V.

Why use generics?

Generics are often required for type safety, but they have more benefits than just allowing your code to run:

- Properly specifying generic types results in better generated code.
- You can use generics to reduce code duplication.

If you intend for a list to contain only strings, you can declare it as `List<String>` (read that as “list of string”). That way you, your fellow programmers, and your tools can detect that assigning a non-string to the list is probably a mistake. Here's an example:

```
x static analysis: error/warning
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // Error
```

Another reason for using generics is to reduce code duplication. Generics let you share a single interface and implementation between many types, while still taking advantage of static analysis. For example, say you create an interface for caching an object:

```
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}
```

You discover that you want a string-specific version of this interface, so you create another interface:

```
abstract class StringCache {
    String getByKey(String key);
    void setByKey(String key, String value);
}
```

Later, you decide you want a number-specific version of this interface... You get the idea.

Generic types can save you the trouble of creating all these interfaces. Instead, you can create a single interface that takes a type parameter:

```
abstract class Cache<T> {
    T getByKey(String key);
    void setByKey(String key, T value);
}
```

In this code, T is the stand-in type. It's a placeholder that you can think of as a type that a developer will define later.

Using collection literals

List, set, and map literals can be parameterized. Parameterized literals are just like the literals you've already seen, except that you add `<type>` (for lists and sets) or `<keyType, valueType>` (for maps) before the opening bracket. Here is an example of using typed literals:

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
    'index.html': 'Homepage',
    'robots.txt': 'Hints for web robots',
    'humans.txt': 'We are people, not machines'
};
```

Using parameterized types with constructors

To specify one or more types when using a constructor, put the types in angle brackets (`<...>`) just after the class name. For example:

```
var nameSet = Set<String>.from(names);
```

The following code creates a map that has integer keys and values of type View:

```
var views = Map<int, View>();
```

Generic collections and the types they contain

Dart generic types are *reified*, which means that they carry their type information around at runtime. For example, you can test the type of a collection:

```
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

Note: In contrast, generics in Java use *erasure*, which means that generic type parameters are removed at runtime. In Java, you can test whether an object is a `List`, but you can't test whether it's a `List<String>`.

Restricting the parameterized type

When implementing a generic type, you might want to limit the types that can be provided as arguments, so that the argument must be a subtype of a particular type. You can do this using `extends`.

A common use case is ensuring that a type is non-nullable by making it a subtype of `Object` (instead of the default, `Object?`).

```
class Foo<T extends Object> {  
    // Any type provided to Foo for T must be non-nullable.  
}
```

You can use `extends` with other types besides `Object`. Here's an example of extending `SomeBaseClass`, so that members of `SomeBaseClass` can be called on objects of type `T`:

```
class Foo<T extends SomeBaseClass> {  
    // Implementation goes here...  
    String toString() => "Instance of 'Foo<$T>'";  
}  
  
class Extender extends SomeBaseClass {...}
```

It's OK to use `SomeBaseClass` or any of its subtypes as the generic argument:

```
var someBaseClassFoo = Foo<SomeBaseClass>();  
var extenderFoo = Foo<Extender>();
```

It's also OK to specify no generic argument:

```
var foo = Foo();  
print(foo); // Instance of 'Foo<SomeBaseClass>'
```

Specifying any non-`SomeBaseClass` type results in an error:

```
x static analysis: error/warning  
var foo = Foo<Object>();
```

Using generic methods

Methods and functions also allow type arguments:

```
T first<T>(List<T> ts) {  
    // Do some initial work or error checking, then...  
    T tmp = ts[0];  
    // Do some additional checking or processing...  
    return tmp;  
}
```

Here the generic type parameter on `first` (`<T>`) allows you to use the type argument `T` in several places:

- In the function's return type (`T`).
- In the type of an argument (`List<T>`).
- In the type of a local variable (`T tmp`).

Libraries and visibility

The `import` and `library` directives can help you create a modular and shareable code base. Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (`_`) are visible only inside the library. *Every Dart app is a library*, even if it doesn't use a `library` directive.

Libraries can be distributed using [packages](#).

i If you're curious why Dart uses underscores instead of access modifier keywords like `public` or `private`, see [SDK issue 33383](#).

Using libraries

Use `import` to specify how a namespace from one library is used in the scope of another library.

For example, Dart web apps generally use the [dart:html](#) library, which they can import like this:

```
import 'dart:html';
```

The only required argument to `import` is a URI specifying the library. For built-in libraries, the URI has the special `dart:` scheme. For other libraries, you can use a file system path or the `package:` scheme. The `package:` scheme specifies libraries provided by a package manager such as the pub tool. For example:

```
import 'package:test/test.dart';
```

i Note: *URI* stands for uniform resource identifier. *URLs* (uniform resource locators) are a common kind of URI.

Specifying a library prefix

If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if `library1` and `library2` both have an `Element` class, then you might have code like this:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// Uses Element from lib1.
Element element1 = Element();

// Uses Element from lib2.
lib2.Element element2 = lib2.Element();
```

Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Lazily loading a library

Deferred loading (also called *lazy loading*) allows a web app to load a library on demand, if and when the library is needed. Here are some cases when you might use deferred loading:

- To reduce a web app's initial startup time.
- To perform A/B testing—trying out alternative implementations of an algorithm, for example.
- To load rarely used functionality, such as optional screens and dialogs.

⚠ **Only dart compile js supports deferred loading.** Flutter and the Dart VM don't support deferred loading. To learn more, see [issue #33118](#) and [issue #27776](#).

To lazily load a library, you must first import it using `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

When you need the library, invoke `loadLibrary()` using the library's identifier.

```
Future<void> greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

In the preceding code, the `await` keyword pauses execution until the library is loaded. For more information about `async` and `await`, see [asynchrony support](#).

You can invoke `loadLibrary()` multiple times on a library without problems. The library is loaded only once.

Keep in mind the following when you use deferred loading:

- A deferred library's constants aren't constants in the importing file. Remember, these constants don't exist until the deferred library is loaded.
- You can't use types from a deferred library in the importing file. Instead, consider moving interface types to a library imported by both the deferred library and the importing file.
- Dart implicitly inserts `loadLibrary()` into the namespace that you define using `deferred as namespace`. The `loadLibrary()` function returns a [Future](#).

The `library` directive

To specify library-level [doc comments](#) or [metadata annotations](#), attach them to a `library` declaration at the start of the file.

```
/// A really great test library.
@TestOn('browser')
library;
```

Implementing libraries

See [Create Library Packages](#) for advice on how to implement a library package, including:

- How to organize library source code.

- How to use the `export` directive.
- When to use the `part` directive.
- How to use conditional imports and exports to implement a library that supports multiple platforms.

Asynchrony support

Dart libraries are full of functions that return [Future](#) or [Stream](#) objects. These functions are *asynchronous*: they return after setting up a possibly time-consuming operation (such as I/O), without waiting for that operation to complete.

The `async` and `await` keywords support asynchronous programming, letting you write asynchronous code that looks similar to synchronous code.

Handling Futures

When you need the result of a completed Future, you have two options:

- Use `async` and `await`, as described here and in the [asynchronous programming codelab](#).
- Use the Future API, as described [in the library tour](#).

Code that uses `async` and `await` is asynchronous, but it looks a lot like synchronous code. For example, here's some code that uses `await` to wait for the result of an asynchronous function:

```
await lookUpVersion();
```

To use `await`, code must be in an `async` function—a function marked as `async`:

```
Future<void> checkVersion() async {
  var version = await lookUpVersion();
  // Do something with version
}
```

Note: Although an `async` function might perform time-consuming operations, it doesn't wait for those operations. Instead, the `async` function executes only until it encounters its first `await` expression. Then it returns a `Future` object, resuming execution only after the `await` expression completes.

Use `try`, `catch`, and `finally` to handle errors and cleanup in code that uses `await`:

```
try {
  version = await lookUpVersion();
} catch (e) {
  // React to inability to look up the version
}
```

You can use `await` multiple times in an `async` function. For example, the following code waits three times for the results of functions:

```
var entrypoint = await findEntryPoint();
var exitCode = await runExecutable(entrypoint, args);
await flushThenExit(exitCode);
```

In `await expression`, the value of `expression` is usually a Future; if it isn't, then the value is automatically wrapped in a Future. This Future object indicates a promise to return an object. The value of `await expression` is that returned object. The `await` expression makes execution pause until that object is available.

If you get a compile-time error when using `await`, make sure `await` is in an `async` function. For example, to use `await` in your app's `main()` function, the body of `main()` must be marked as `async`:

```
void main() async {  
  checkVersion();  
  print('In main: version is ${await lookUpVersion()}');  
}
```

Note: The preceding example uses an `async` function (`checkVersion()`) without waiting for a result—a practice that can cause problems if the code assumes that the function has finished executing. To avoid this problem, use the [unawaited_futures linter rule](#).

For an interactive introduction to using futures, `async`, and `await`, see the [asynchronous programming codelab](#).

Declaring async functions

An `async` function is a function whose body is marked with the `async` modifier.

Adding the `async` keyword to a function makes it return a Future. For example, consider this synchronous function, which returns a String:

```
String lookUpVersion() => '1.0.0';
```

If you change it to be an `async` function—for example, because a future implementation will be time consuming—the returned value is a Future:

```
Future<String> lookUpVersion() async => '1.0.0';
```

Note that the function's body doesn't need to use the Future API. Dart creates the Future object if necessary. If your function doesn't return a useful value, make its return type `Future<void>`.

For an interactive introduction to using futures, `async`, and `await`, see the [asynchronous programming codelab](#).

Handling Streams

When you need to get values from a Stream, you have two options:

- Use `async` and an *asynchronous for loop* (`await for`).
- Use the Stream API, as described [in the library tour](#).

Note: Before using `await for`, be sure that it makes the code clearer and that you really do want to wait for all of the stream's results. For example, you usually should **not** use `await for` for UI event listeners, because UI frameworks send endless streams of events.

An asynchronous for loop has the following form:

```
await for (varOrType identifier in expression) {  
  // Executes each time the stream emits a value.  
}
```

The value of `expression` must have type Stream. Execution proceeds as follows:

1. Wait until the stream emits a value.
2. Execute the body of the for loop, with the variable set to that emitted value.
3. Repeat 1 and 2 until the stream is closed.

To stop listening to the stream, you can use a `break` or `return` statement, which breaks out of the for loop and unsubscribes from the stream.

If you get a compile-time error when implementing an asynchronous for loop, make sure the `await for` is in an `async` function. For example, to use an asynchronous for loop in your app's `main()` function, the body of `main()` must be marked as `async`:

```
void main() async {  
  // ...  
  await for (final request in requestServer) {  
    handleRequest(request);  
  }  
  // ...  
}
```

For more information about asynchronous programming, in general, see the [dart:async](#) section of the library tour.

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:

```
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:

```
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*`:

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

Callable classes

To allow an instance of your Dart class to be called like a function, implement the `call()` method.

The `call()` method allows any class that defines it to emulate a function. This method supports the same functionality as normal [functions](#) such as parameters and return types.

In the following example, the `WannabeFunction` class defines a `call()` function that takes three strings and concatenates them, separating each with a space, and appending an exclamation. Click **Run** to execute the code.



The screenshot shows an IDE window with a tab labeled 'Dart'. At the top right, there are buttons for 'Install SDK', 'Format', 'Reset', and a 'Run' button with a play icon. Below the code editor, there is a 'Console' panel. The code in the editor is as follows:

```
xxxxxxxxx
1
class WannabeFunction {
2
    String call(String a, String b, String c) => '$a $b $c!';
3
}
4

5
var wf = WannabeFunction();
```

The console panel shows the output 'xxxxxxxxx' and a status 'no issues' at the bottom right.

Isolates

Most computers, even on mobile platforms, have multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code.

Instead of threads, all Dart code runs inside of *isolates*. Each Dart isolate uses a single thread of execution and shares no mutable objects with other isolates. Spinning up multiple isolates creates multiple threads of execution. This enables multi-threading without its primary drawback, [race conditions](#).

For more information, see the following:

- [Concurrency in Dart](#)
- [dart:isolate API reference](#), including [Isolate.spawn\(\)](#) and [TransferableTypedData](#)
- [Background parsing](#) cookbook on the Flutter site
- [Isolate sample app](#)

Typedefs

A type alias—often called a *typedef* because it's declared with the keyword `typedef`—is a concise way to refer to a type. Here's an example of declaring and using a type alias named `IntList`:

```
typedef IntList = List<int>;
IntList il = [1, 2, 3];
```

A type alias can have type parameters:

```
typedef ListMapper<X> = Map<X, List<X>>;
Map<String, List<String>> m1 = {}; // Verbose.
ListMapper<String> m2 = {}; // Same thing but shorter and clearer.
```

🔗 **Version note:** Before 2.13, typedefs were restricted to function types. Using the new typedefs requires a [language version](#) of at least 2.13.

We recommend using [inline function types](#) instead of typedefs for functions, in most situations. However, function typedefs can still be useful:

```
typedef Compare<T> = int Function(T a, T b);

int sort(int a, int b) => a - b;

void main() {
  assert(sort is Compare<int>); // True!
}
```

Metadata

Use metadata to give additional information about your code. A metadata annotation begins with the character `@`, followed by either a reference to a compile-time constant (such as `deprecated`) or a call to a constant constructor.

Three annotations are available to all Dart code: `@Deprecated`, `@deprecated`, and `@override`. For examples of using `@override`, see [Extending a class](#). Here's an example of using the `@Deprecated` annotation:

```
class Television {
  /// Use [turnOn] to turn the power on instead.
  @Deprecated('Use turnOn instead')
  void activate() {
    turnOn();
  }

  /// Turns the TV's power on.
  void turnOn() {...}
  // ...
}
```

You can define your own metadata annotations. Here's an example of defining a `@Todo` annotation that takes two arguments:

```
class Todo {
  final String who;
  final String what;

  const Todo(this.who, this.what);
}
```

And here's an example of using that `@Todo` annotation:

```
@Todo('Dash', 'Implement this function')
void doSomething() {
  print('Do something');
}
```

Metadata can appear before a library, class, typedef, type parameter, constructor, factory, function, field, parameter, or variable declaration and before an import or export directive. You can retrieve metadata at runtime using reflection.

Comments

Dart supports single-line comments, multi-line comments, and documentation comments.

Single-line comments

A single-line comment begins with `//`. Everything between `//` and the end of line is ignored by the Dart compiler.

```

void main() {
  // TODO: refactor into an AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}

```

Multi-line comments

A multi-line comment begins with `/*` and ends with `*/`. Everything between `/*` and `*/` is ignored by the Dart compiler (unless the comment is a documentation comment; see the next section). Multi-line comments can nest.

```

void main() {
  /*
   * This is a lot of work. Consider raising chickens.

   Llama larry = Llama();
   larry.feed();
   larry.exercise();
   larry.clean();
  */
}

```

Documentation comments

Documentation comments are multi-line or single-line comments that begin with `///` or `/**`. Using `///` on consecutive lines has the same effect as a multi-line doc comment.

Inside a documentation comment, the analyzer ignores all text unless it is enclosed in brackets. Using brackets, you can refer to classes, methods, fields, top-level variables, functions, and parameters. The names in brackets are resolved in the lexical scope of the documented program element.

Here is an example of documentation comments with references to other classes and arguments:

```

/// A domesticated South American camelid (Lama glama).
///
/// Andean cultures have used llamas as meat and pack
/// animals since pre-Hispanic times.
///
/// Just like any other animal, llamas need to eat,
/// so don't forget to [feed] them some [Food].
class Llama {
  String? name;

  /// Feeds your llama [food].
  ///
  /// The typical llama eats one bale of hay per week.
  void feed(Food food) {
    // ...
  }

  /// Exercises your llama with an [activity] for
  /// [timeLimit] minutes.
  void exercise(Activity activity, int timeLimit) {
    // ...
  }
}

```

In the class's generated documentation, `[feed]` becomes a link to the docs for the `feed` method, and `[Food]` becomes a link to the docs for the `Food` class.

To parse Dart code and generate HTML documentation, you can use Dart's documentation generation tool, [dart doc](#). For an example of generated documentation, see the [Dart API documentation](#). For advice on how to structure your comments, see [Effective Dart: Documentation](#).

Summary

This page summarized the commonly used features in the Dart language. More features are being implemented, but we expect that they won't break existing code. For more information, see the [Dart language specification](#) and [Effective Dart](#).

To learn more about Dart's core libraries, see [A Tour of the Dart Libraries](#).