Table of Contents

Chapter 1: Getting started with dart

Remarks

Dart is an open-source, class-based, optionally-typed programming language for building web applications--on both the client and server--created by Google. Dart's design goals are:

- Create a structured yet flexible language for web programming.
- Make Dart feel familiar and natural to programmers and thus easy to learn.
- Ensure that Dart delivers high performance on all modern web browsers and environments ranging from small handheld devices to server-side execution.

Dart targets a wide range of development scenarios, from a one-person project without much structure to a large-scale project needing formal types in the code to state programmer intent.

To support this wide range of projects, Dart provides the following features and tools:

- **Optional types:** this means you can start coding without types and add them later as needed.
- **Isolates:** concurrent programming on server and client
- **Easy DOM access:** using CSS selectors (the same way that jQuery does it)
- **Dart IDE Tools:** Dart plugins exist for many commonly used IDEs, Ex: WebStorm.
- **Dartium:** a build of the Chromium Web Browser with a built-in Dart Virtual Machine

Links

- The Dart Homepage
- Official Dart News & Updates
- The Dartosphere - A collection of recent Dart blog posts
- Dartisans Dartisans community on Google+
- Dart Web Development - Google Groups Page
- Dart Language Misc - Google Groups Page
- DartLang sub-Reddit

Documentation

- Tour of the Dart Language
- Tour of the Dart Libraries
- Dart Code samples
- Dart API Reference

FAQ

- Frequently Asked Questions

Versions

Version	Release Date
1.22.1	2017-02-22
1.22.0	2017-02-14
1.21.1	2016-01-13
1.21.0	2016-12-07
1.20.1	2016-10-13
1.20.0	2016-10-11
1.19.1	2016-09-07
1.19.0	2016-08-26
1.18.1	2016-08-02
1.18.0	2016-07-27
1.17.1	2016-06-10
1.17.0	2016-06-06
1.16.1	2016-05-23
1.16.0	2016-04-26
1.15.0	2016-03-09
1.14.2	2016-02-09
1.14.1	2016-02-03
1.14.0	2016-01-28
1.13.2	2016-01-05
1.13.1	2015-12-17
1.13.0	2015-11-18

Version	Release Date
1.12.2	2015-10-21
1.12.1	2015-09-08
1.12.0	2015-08-31
1.11.3	2015-08-03
1.11.1	2015-07-02
1.11.0	2015-06-24
1.10.1	2015-05-11
1.10.0	2015-04-24
1.9.3	2015-04-13
1.9.1	2015-03-25
1.8.5	2015-01-13
1.8.3	2014-12-01
1.8.0	2014-11-27
1.7.2	2014-10-14
1.6.0	2014-08-27
1.5.8	2014-07-29
1.5.3	2014-07-03
1.5.2	2014-07-02
1.5.1	2014-06-24
1.4.3	2014-06-16
1.4.2	2014-05-27
1.4.0	2014-05-20
1.3.6	2014-04-30
1.3.3	2014-04-16
1.3.0	2014-04-08

Version	Release Date
1.2.0	2014-02-25
1.1.3	2014-02-06
1.1.1	2014-01-15
1.0.0.10_r30798	2013-12-02
1.0.0.3_r30188	2013-11-12
0.8.10.10_r30107	2013-11-08
0.8.10.6_r30036	2013-11-07
0.8.10.3_r29803	2013-11-04

Examples

Installation or Setup

The Dart SDK includes everything you need to write and run Dart code: VM, libraries, analyzer, package manager, doc generator, formatter, debugger, and more. If you are doing web development, you will also need Dartium.

Automated installation and updates

- Installing Dart on Windows
- Installing Dart on Mac
- Installing Dart on Linux

Manual install

You can also manually install any version of the SDK.

Hello, World!

Create a new file named `hello_world.dart` with the following content:

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

In the terminal, navigate to the directory containing the file `hello_world.dart` and type the following:

```
dart hello_world.dart
```

Hit `enter` to display `Hello, World!` in the terminal window.

Http Request

Html

```
<img id="cats"></img>
```

Dart

```
import 'dart:html';

/// Stores the image in [blob] in the [ImageElement] of the given [selector].
void setImage(selector, blob) {
  FileReader reader = new FileReader();
  reader.onLoad.listen((fe) {
    ImageElement image = document.querySelector(selector);
    image.src = reader.result;
  });
  reader.readAsDataUrl(blob);
}

main() async {
  var url = "https://upload.wikimedia.org/wikipedia/commons/2/28/Tortoiseshell_she-cat.JPG";

  // Initiates a request and asynchronously waits for the result.
  var request = await HttpRequest.request(url, responseType: 'blob');
  var blob = request.response;
  setImage("#cats", blob);
}
```

Example

see Example on https://dartpad.dartlang.org/a0e092983f63a40b0b716989cac6969a

Getters and Setters

```
void main() {
  var cat = new Cat();

  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? true
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? false
  print("Feed cat.");
  cat.isHungry = false;
  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? false
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? true
}

class Cat {
  bool _isHungry = true;

  bool get isCuddly => !_isHungry;
```

```
  bool get isHungry => _isHungry;
  bool set isHungry(bool hungry) => this._isHungry = hungry;
}
```

Dart class getters and setters allow APIs to encapsulate object state changes.

See dartpad example here: https://dartpad.dartlang.org/c25af60ca18a192b84af6990f3313233

Read Getting started with dart online: https://riptutorial.com/dart/topic/843/getting-started-with-dart

Chapter 2: Asynchronous Programming

Examples

Returning a Future using a Completer

```
Future<Results> costlyQuery() {
  var completer = new Completer();

  database.query("SELECT * FROM giant_table", (results) {
    // when complete
    completer.complete(results);
  }, (error) {
    completer.completeException(error);
  });

  // this returns essentially immediately,
  // before query is finished
  return completer.future;
}
```

Async and Await

```
import 'dart:async';

Future main() async {
  var value = await _waitForValue();
  print("Here is the value: $value");
  //since _waitForValue() returns immediately if you un it without await you won't get the
result
  var errorValue = "not finished yet";
  _waitForValue();
  print("Here is the error value: $value");// not finished yet
}

Future<int> _waitForValue() => new Future((){

  var n = 100000000;

  // Do some long process
  for (var i = 1; i <= n; i++) {
    // Print out progress:
    if ([n / 2, n / 4, n / 10, n / 20].contains(i)) {
      print("Not done yet...");
    }

    // Return value when done.
    if (i == n) {
      print("Done.");
      return i;
    }
  }
});
```

See example on Dartpad: https://dartpad.dartlang.org/11d189b51e0f2680793ab3e16e53613c

Converting callbacks to Futures

Dart has a robust async library, with Future, Stream, and more. However, sometimes you might run into an asynchronous API that uses *callbacks* instead of *Futures*. To bridge the gap between callbacks and Futures, Dart offers the *Completer* class. You can use a Completer to convert a callback into a Future.

Completers are great for bridging a callback-based API with a Future-based API. For example, suppose your database driver doesn't use Futures, but you need to return a Future. Try this code:

```
// A good use of a Completer.

Future doStuff() {
  Completer completer = new Completer();
  runDatabaseQuery(sql, (results) {
    completer.complete(results);
  });
  return completer.future;
}
```

If you are using an API that already returns a Future, you do not need to use a Completer.

Read Asynchronous Programming online: https://riptutorial.com/dart/topic/2520/asynchronous-programming

Chapter 3: Classes

Examples

Creating a class

Classes can be created as follow:

```
class InputField {
  int maxLength;
  String name;
}
```

The class can be instantiated using the `new` keyword after which the field values will be null by default.

```
var field = new InputField();
```

Field values can then be accessed:

```
// this will trigger the setter
field.name = "fieldname";

// this will trigger the getter
print(field.name);
```

Members

A class can have members.

Instance variables can be declared with/without type annotations, and optionally initialized. Uninitialised members have the value of `null`, unless set to another value by the constructor.

```
class Foo {
  var member1;
  int member2;
  String member3 = "Hello world!";
}
```

Class variables are declared using the `static` keyword.

```
class Bar {
  static var member4;
  static String member5;
  static int member6 = 42;
}
```

If a method takes no arguments, is fast, returns a value, and doesn't have visible side-effects, then

a getter method can be used:

```
class Foo {
  String get bar {
    var result;
    // ...
    return result;
  }
}
```

Getters never take arguments, so the parentheses for the (empty) parameter list are omitted both for declaring getters, as above, and for calling them, like so:

```
main() {
  var foo = new Foo();
  print(foo.bar); // prints "bar"
}
```

There are also setter methods, which must take exactly one argument:

```
class Foo {
  String _bar;

  String get bar => _bar;

  void set bar(String value) {
    _bar = value;
  }
}
```

The syntax for calling a setter is the same as variable assignment:

```
main() {
  var foo = new Foo();
  foo.bar = "this is calling a setter method";
}
```

Constructors

A class constructor must have the same name as its class.

Let's create a constructor for a class Person:

```
class Person {
  String name;
  String gender;
  int age;

  Person(this.name, this.gender, this.age);
}
```

The example above is a simpler, better way of defining the constructor than the following way, which is also possible:

```
class Person {
  String name;
  String gender;
  int age;

  Person(String name, String gender, int age) {
    this.name = name;
    this.gender = gender;
    this.age = age;
  }
}
```

Now you can create an instance of Person like this:

```
var alice = new Person('Alice', 'female', 21);
```

Read Classes online: https://riptutorial.com/dart/topic/1511/classes

Chapter 4: Collections

Examples

Creating a new List

Lists can be created in multiple ways.

The recommended way is to use a `List` literal:

```
var vegetables = ['broccoli', 'cabbage'];
```

The `List` constructor can be used as well:

```
var fruits = new List();
```

If you prefer stronger typing, you can also supply a type parameter in one of the following ways:

```
var fruits = <String>['apples', 'oranges'];
var fruits = new List<String>();
```

For creating a small growable list, either empty or containing some known initial values, the literal form is preferred. There are specialized constructors for other kinds of lists:

```
var fixedLengthList1 = new List(8);
var fixedLengthList2 = new List.filled(8, "initial text");
var computedValues = new List.generate(8, (n) => "x" * n);
var fromIterable = new List<String>.from(computedValues.getRange(2, 5));
```

See also the Effective Dart style guide about collections.

Creating a new Set

Sets can be created via the constructor:

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
```

Creating a new Map

Maps can be created in multiple ways.

Using the constructor, you can create a new map as follow:

```
var searchTerms = new Map();
```

Types for the key and value can also be defined using generics:

```
var nobleGases = new Map<int, String>();
var nobleGases = <int, String>{};
```

Maps can otherwise be created using the map literal:

```
var map = {
    "key1": "value1",
    "key2": "value2"
};
```

Map each element in the collection.

All collection objects contain a `map` method that takes a `Function` as an argument, which must take a single argument. This returns an `Iterable` backed by the collection. When the `Iterable` is iterated, each step calls the function with a new element of the collection, and the result of the call becomes the next element of the iteration.

You can turn an `Iterable` into a collection again by using the `Iterable.toSet()` or `Iterable.toList()` methods, or by using a collection constructor which takes an iterable like `Queue.from` or `List.from`.

Example:

```
main() {
  var cats = [
    'Abyssinian',
    'Scottish Fold',
    'Domestic Shorthair'
  ];

  print(cats); // [Abyssinian, Scottish Fold, Domestic Shorthair]

  var catsInReverse =
  cats.map((String cat) {
    return new String.fromCharCodes(cat.codeUnits.reversed);
  })
  .toList(); // [nainissybA, dloF hsittocS, riahtrohS citsemoD]

  print(catsInReverse);
}
```

See dartpad example here: https://dartpad.dartlang.org/a18367ff767f172b34ff03c7008a6fa1

Filter a list

Dart allows to easily filter a list using `where`.

```
var fruits = ['apples', 'oranges', 'bananas'];
fruits.where((f) => f.startsWith('a')).toList(); //apples
```

Of course you can use some AND or OR operators in your where clause.

Chapter 5: Comments

Syntax

- // Single-line comment
- /* Multi-line/In-line comment */
- /// Dartdoc comment

Remarks

It is good practice to add comments to your code to explain why something is done or to explain what something does. This helps any future readers of your code to more easily understand your code.

Related topic(s) not on StackOverflow:

- Effective Dart: Documentation

Examples

End of Line Comment

Everything to the right of `//` in the same line is commented.

```
int i = 0; // Commented out text
```

Multi-Line Comment

Everything between `/*` and `*/` is commented.

```
void main() {
  for (int i = 0; i < 5; i++) {
    /* This is commented, and
    will not affect code */
    print('hello ${i + 1}');
  }
}
```

Documentation using Dartdoc

Using a doc comment instead of a regular comment enables dartdoc to find it and generate documentation for it.

```
/// The number of characters in this chunk when unsplit.
int get length => ...
```

You are allowed to use most markdown formatting in your doc comments and dartdoc will process it accordingly using the markdown package.

```
/// This is a paragraph of regular text.
///
/// This sentence has *two* _emphasized_ words (i.e. italics) and **two**
///___strong__ones (bold).
///
/// A blank line creates another separate paragraph. It has some `inline code`
/// delimited using backticks.
///
/// * Unordered lists.
/// * Look like ASCII bullet lists.
/// * You can also use `-` or `+`.
///
/// Links can be:
///
/// * http://www.just-a-bare-url.com
/// * [with the URL inline](http://google.com)
/// * [or separated out][ref link]
///
/// [ref link]: http://google.com
///
/// # A Header
///
/// ## A subheader
```

Chapter 6: Control Flow

Examples

If Else

Dart has If Else:

```
if (year >= 2001) {
  print('21st century');
} else if (year >= 1901) {
  print('20th century');
} else {
  print('We Must Go Back!');
}
```

Dart also has a ternary `if` operator:

```
var foo = true;
print(foo ? 'Foo' : 'Bar'); // Displays "Foo".
```

While Loop

While loops and do while loops are allowed in Dart:

```
while(peopleAreClapping()) {
  playSongs();
}
```

and:

```
do {
  processRequest();
} while(stillRunning());
```

Loops can be terminated using a break:

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

You can skip iterations in a loop using continue:

```
for (var i = 0; i < bigNumber; i++) {
  if (i.isEven){
    continue;
  }
  doSomething();
```

```
}
```

For Loop

Two types of for loops are allowed:

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

and:

```
for (var object in flybyObjects) {
  print(object);
}
```

The `for-in` loop is convenient when simply iterating over an `Iterable` collection. There is also a `forEach` method that you can call on `Iterable` objects that behaves like `for-in`:

```
flybyObjects.forEach((object) => print(object));
```

or, more concisely:

```
flybyObjects.forEach(print);
```

Switch Case

Dart has a switch case which can be used instead of long if-else statements:

```
var command = 'OPEN';

switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'OPEN':
    executeOpen();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'UNSURE':
    // missing break statement means this case will fall through
    // to the next statement, in this case the default case
  default:
    executeUnknown();
}
```

You can only compare integer, string, or compile-time constants. The compared objects must be instances of the same class (and not of any of its subtypes), and the class must not override ==.

One surprising aspect of switch in Dart is that non-empty case clauses must end with break, or less commonly, continue, throw, or return. That is, non-empty case clauses cannot fall through. You must explicitly end a non-empty case clause, usually with a break. You will get a static warning if you omit break, continue, throw, or return, and the code will error at that location at runtime.

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break causes an exception to be thrown!!

  case 'CLOSED': // Empty case falls through
  case 'LOCKED':
    executeClosed();
    break;
}
```

If you want fall-through in a non-empty `case`, you can use `continue` and a label:

```
      var command = 'OPEN';
      switch (command) {
        case 'OPEN':
          executeOpen();
          continue locked;
locked: case 'LOCKED':
          executeClosed();
          break;
      }
```

Chapter 7: Enums

Examples

Basic usage

```
enum Fruit {
  apple, banana
}

main() {
  var a = Fruit.apple;
  switch (a) {
    case Fruit.apple:
      print('it is an apple');
      break;
  }

  // get all the values of the enums
  for (List<Fruit> value in Fruit.values) {
    print(value);
  }

  // get the second value
  print(Fruit.values[1]);
}
```

Read Enums online: https://riptutorial.com/dart/topic/5107/enums

Chapter 8: Exceptions

Remarks

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 do not declare which exceptions they might throw, and you are not 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.

Examples

Custom exception

```
class CustomException implements Exception {
  String cause;
  CustomException(this.cause);
}

void main() {
  try {
    throwException();
  } on CustomException {
    print("custom exception is been obtained");
  }
}

throwException() {
  throw new CustomException('This is my first custom exception');
}
```

Chapter 9: Functions

Remarks

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.

Examples

Functions with named parameters

When defining a function, use {param1, param2, …} to specify named parameters:

```
void enableFlags({bool bold, bool hidden}) {
  // ...
}
```

When calling a function, you can specify named parameters using paramName: value

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

Function scoping

Dart functions may also be declared anonymously or nested. For example, to create a nested function, just open a new function block within an existing function block

```
void outerFunction() {

    bool innerFunction() {
        /// Does stuff
    }
}
```

The function `innerFunction` may now be used inside, and only inside, `outerFunction`. No other other functions has access to it.

Functions in Dart may also be declared anonymously, which is commonly used as function arguments. A common example is the `sort` method of `List` object. This method takes an optional argument with the following signature:

```
int compare(E a, E b)
```

The documentation states that the function must return `0` if the `a` and `b` are equal. It returns `-1` if `a < b` and `1` if `a > b`.

Knowing this, we can sort a list of integers using an anonymous function.

```
List<int> numbers = [4,1,3,5,7];

numbers.sort((int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
});
```

Anonymous function may also be bound to identifiers like so:

```
Function intSorter = (int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
}
```

and used as an ordinary variable.

```
numbers.sort(intSorter);
```

Chapter 10: Libraries

Remarks

The `import` and `library` directives can help you create a modular and shareable code base. Every Dart app is a `library`, even if it doesn't use a library directive. Libraries can be distributed using packages. See Pub Package and Asset Manager for information about pub, a package manager included in the SDK.

Examples

Using libraries

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

```
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 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

Libraries and visibility

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.

If you for example have class A in a separate library file (eg, `other.dart`), such as:

```
library other;

class A {
  int _private = 0;

  testA() {
    print('int value: $_private');  // 0
    _private = 5;
    print('int value: $_private'); // 5
  }
}
```

and then import it into your main app, such as:

```
import 'other.dart';

void main() {
  var b = new B();
  b.testB();
}

class B extends A {
  String _private;

  testB() {
    _private = 'Hello';
    print('String value: $_private'); // Hello
    testA();
    print('String value: $_private'); // Hello
  }
}
```

You get the expected output:

```
String value: Hello
int value: 0
int value: 5
String value: Hello
```

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;
// ...
var element1 = new Element(); // Uses Element from lib1.
var element2 =
    new lib2.Element();       // Uses Element from lib2.
```

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 and bar.
import 'package:lib1/lib1.dart' show foo, bar;

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

Lazily loading a library

Deferred loading (also called lazy loading) allows an application to load a library on demand, if and when it’s needed. To lazily load a library, you must first import it using deferred as.

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

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

```
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 more examples here asynchrony support or visit the asynchrony support part of the language tour.

Chapter 11: List Filters

Introduction

Dart filters lists through the `List.where` and `List.retainWhere` methods. The `where` function takes one argument: a boolean function that is applied to each element of the list. If the function evaluates to `true` then the list element is retained; if the function evaluates to `false`, the element is removed.

Calling `theList.retainWhere(foo)` is practically equivalent to setting `theList = theList.where(foo)`.

Examples

Filtering a list of integers

```
[-1, 0, 2, 4, 7, 9].where((x) => x > 2) --> [4, 7, 9]
```

Chapter 12: Pub

Remarks

When you install the Dart SDK, one of the tools that you get is pub. The pub tool provides commands for a variety of purposes. One command installs packages, another starts up an HTTP server for testing, another prepares your app for deployment, and another publishes your package to pub.dartlang.org. You can access the pub commands either through an IDE, such as WebStorm, or at the command line.

For an overview of these commands, see Pub Commands.

Examples

pub build

Use pub build when you're ready to deploy your web app. When you run pub build, it generates the assets for the current package and all of its dependencies, putting them into new directory named build.

To use `pub build`, just run it in your package's root directory. For example:

```
$ cd ~/dart/helloworld
$ pub build
Building helloworld......
Built 5 files!
```

pub serve

This command starts up a development server, or dev server, for your Dart web app. The dev server is an HTTP server on localhost that serves up your web app's assets.

Start the dev server from the directory that contains your web app's `pubspec.yaml` file:

```
$ cd ~/dart/helloworld
$ pub serve
Serving helloworld on http://localhost:8080
```

Chapter 13: Regular Expressions

Syntax

- var regExp = RegExp(r'^(.*)$', multiLine: true, caseSensitive: false);

Parameters

Parameter	Details
`String source`	The regular expression as a `String`
`{bool multiline}`	Whether this is a multiline regular expression. (matches `^` and `$` at the beginning and end of each line individually not the whole String)
`{bool caseSensitive}`	If the expression is case sensitive

Remarks

Dart regular expressions have the same syntax and semantics as JavaScript regular expressions. See http://ecma-international.org/ecma-262/5.1/#sec-15.10 for the specification of JavaScript regular expressions.

This means that any JavaScript resource you find about Regular Expressions online applies to dart.

Examples

Create and use a Regular Expression

```
var regExp = new RegExp(r"(\w+)");
var str = "Parse my string";
Iterable<Match> matches = regExp.allMatches(str);
```

It's a good idea to use "raw strings" (prefix with `r`) when writing regular expressions so you can use unescaped backslashes in your expression.

Read Regular Expressions online: https://riptutorial.com/dart/topic/3624/regular-expressions

Chapter 14: Strings

Examples

Concatenation and interpolation

You can use the plus (+) operator to concatenate strings:

```
'Dart ' + 'is ' + 'fun!'; // 'Dart is fun!'
```

You can also use adjacent string literals for concatenation:

```
'Dart ' 'is ' 'fun!';    // 'Dart is fun!'
```

You can use `${}` to interpolate the value of Dart expressions within strings. The curly braces can be omitted when evaluating identifiers:

```
var text = 'dartlang';
'$text has ${text.length} letters'; // 'dartlang has 8 letters'
```

Valid strings

A string can be either single or multiline. Single line strings are written using matching single or double quotes, and multiline strings are written using triple quotes. The following are all valid Dart strings:

```
'Single quotes';
"Double quotes";
'Double quotes in "single" quotes';
"Single quotes in 'double' quotes";

'''A
multiline
string''';

"""
Another
multiline
string""";
```

Building from parts

Programmatically generating a String is best accomplished with a StringBuffer. A StringBuffer doesn't generate a new String object until `toString()` is called.

```
var sb = new StringBuffer();

sb.write("Use a StringBuffer");
```

```
sb.writeAll(["for ", "efficient ", "string ", "creation "]);
sb.write("if you are ")
sb.write("building lots of strings");

// or you can use method cascades:

sb
  ..write("Use a StringBuffer")
  ..writeAll(["for ", "efficient ", "string ", "creation "])
  ..write("if you are ")
  ..write("building lots of strings");

var fullString = sb.toString();

print(fullString);
// Use a StringBufferfor efficient string creation if you are building lots of strings

sb.clear(); // all gone!
```