Chapter II

All About Dart Language

I. Resources for Dart (Read First)

https://dart.dev/guides/language/language-tour

https://dartpad.dev/

II. 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. All objects inherit from the Object class.

- 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. When you want to explicitly say that no type is expected, use the special type dynamic.

- Dart supports generic types, like List<int> (a list of integers) or List<dynamic> (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.

- 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.

III.Keywords & Identifiers

Keywords

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

abstract 2	dynamic 2	implements 2	show 1
as 2	else	import 2	static 2
assert	enum	in	super
async 1	export 2	interface 2	switch
await 3	extends	is	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	rethrow	while
deferred 2	hide 1	return	with
do	if	set 2	yield 3

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. To simplify the task of porting JavaScript code to Dart, 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 newer, limited reserved words related to the asynchrony support that was added after Dart's 1.0 release. 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.

Identifiers

An identifier is a name that is assigned by the user for a program element such as variable, type, template, class, function or namespace.

Identifiers are used to identify a program element in the code.

Eg.

```
int meaningOfLife = 42;
double valueOfPi  = 3.141592;
bool visible      = true;
```

For creating to store a value in memory , we use variables and Dart language has special support for the following types:

- **numbers**
- **strings**
- **booleans**
- **lists (also known as arrays)**
- **sets**
- **maps**
- **runes (for expressing Unicode characters in a string)**
- **symbols**

Note:

Every variable in Dart refers to an object—an instance of a class—you can usually use constructors to initialize variables.

IV.Default Value

Uninitialized variables have an initial value of null. Even variables with numeric types are initially null, because numbers—like everything else in Dart—are objects.

Eg.

```
int number1;
print(number1)
```

V. Final & Constant Value

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.) A final top-level or class variable is initialized the first time it's used.

Eg.

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

```
final String nickname = 'Bobby';
name = 'Alice'; // Error: a final variable can't be set again

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

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

foo = [1, 2, 3]; // Was const []
baz = [42]; // Error: Constant variables can't be assigned
```

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`.

VI.Operator

Dart defines the operators shown in the following table. You can override many of these operators, as described in Overridable operators.

Description	Operator
unary postfix	`expr++ expr-- () [] . ?.`
unary prefix	`-expr !expr ~expr ++expr --expr`
multiplicative	`* / % ~/`
additive	`+ -`
shift	`<< >> >>>`
bitwise AND	`&`
bitwise XOR	`^`
bitwise OR	`|`
relational and type test	`>= > <= < as is is!`

equality	`==` `!=`
logical AND	`&&`
logical OR	`||`
if null	`??`
conditional	*`expr1 ? expr2 : expr3`*
cascade	`..`
assignment	`=` `*=` `/=` `+=` `-=` `&=` `^=` *etc.*

Arithmetic operators

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

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

Equality and relational operators

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

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

Type test operators

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

Operator	Meaning
as	Typecast (also used to specify library prefixes)
is	True if the object has the specified type
is!	False if the object has the specified type

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.

`=`	`-=`	`/=`	`%=`	`>>=`	`^=`
`+=`	`*=`	`~/=`	`<<=`	`&=`	`|=`

Here's how compound assignment operators work:

	Compound assignment	**Equivalent expression**
For an operator *op*:	`a op= b`	`a = a op b`
Example:	`a += b`	`a = a + b`

The following example uses assignment and compound assignment operators:

```
var a = 2; // Assign using =
a *= 3; // Assign and multiply: a = a * 3
```

Logical operators

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

Operator	**Meaning**
`!expr`	inverts the following expression (changes false to true, and vice versa)
`||`	logical OR
`&&`	logical AND

Eg

```
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
`&`	AND
`|`	OR
`^`	XOR
`~expr`	Unary bitwise complement (0s become 1s; 1s become 0s)
`<<`	Shift left
`>>`	Shift right

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

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

VII.Other operators

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

Operator	Name	Meaning
`()`	Function application	Represents a function call
`[]`	List access	Refers to the value at the specified index in the list
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)

VIII.Identifier In Detail

int

Integer values no larger than 64 bits, depending on the platform. On the Dart VM, values can be from -263 to 263 - 1. Dart that's compiled to JavaScript uses JavaScript numbers, allowing values from -253 to 253 - 1.

double

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

Eg.
```
var x = 1;

var hex = 0xDEADBEEF;
```

```
var y = 1.1;

var exponents = 1.42e5;

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

```
//Type Conversion

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


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


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

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);

const msPerSecond = 1000;

const secondsUntilRetry = 5;

const msUntilRetry = secondsUntilRetry * msPerSecond;
```

Strings

A Dart string is a sequence of **UTF-16** code units. You can use either single or double quotes to create a string:

Eg.

```
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.

Eg.

```
var s = 'string interpolation';
```

```
      var num = 3;
      var sInterpolated = 'Dart has $s, which is very handy.' ;
      var sInterpolated2 = 'Number has $num' ;
      var sInterpolated3 = 'having operation ${num * 3}'  ;

      print(sInterpolated);
      print(sInterpolated2);
      print(sInterpolated3);


// String Concating
Eg.
var sConcated = 'String '
   'concatenation'
   " works even over line breaks.";

var sConcated2 = 'The + operator ' + 'works, as well.';

//String Multi Line

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

var s2 = """This is also a
multi-line string.""";    //Triple double quote


//Raw String  : Start with small r

 var str = "This is how it work\t simple treated string";
 var rawStr = r"This is how it work\t raw string";


print(str);

print(rawStr);
```

Booleans (the boolean literals true and false are used)

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 and use zero-based indexing.eg

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

var constantList = const [1, 2, 3];
```

//The spread operator (...)

//Insert all the elements of a list into another list:

```
var list = [1, 2, 3];
var list2 = [0, ...list];
```

//using a null-aware spread operator (...?):

```
var list;
var list2 = [0, ...?list];
```

//if in list , conditionally

```
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:

//For in the list iterated

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

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.

Eg.

```
var names = <String>{};
// Set<String> names = {}; // This works, too.

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

 //Set  <String> halogens = {'fluorine', 'chlorine', 'bromine', 'iodine',
'astatine'};

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

final constantSet = const {
  'fluorine',
  'chlorine',
  'bromine',
  'iodine',
  'astatine',
};
// constantSet.add('helium'); // Uncommenting this causes an error.
```

Note:

As of Dart 2.3, sets support spread operators (... and ...?) and collection ifs and fors, just like lists do.

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.

Eg.

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

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

// gifts has the type Map<String, String> and
//nobleGases has the type Map<int, String>.



//Creating the same objects using a Map constructor:

var myGifts = Map();
myGifts['first'] = 'bag';
myGifts['second'] = 'turtledoves';
myGifts['fifth'] = 'golden rings';

var myNobleGases = Map();
myNobleGases[2] = 'helium';
myNobleGases[10] = 'neon';
myNobleGases[18] = `argon';

 //  for a key that isn't in a map, you get a null in return:
print(anotherGifts['third'] )
print( myNobleGases[3])


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

//Creating a map with a compile-time constant, add const before the map
literal:

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

// constantMap[2] = 'Helium'; // Uncommenting this causes an error.
```

Runes

In Dart, runes are the **UTF-32** code points of a string.

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 32-bit Unicode values 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{1f600}.

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.

Eg.

```
var mysymbol = 1;

print(#mysymbol);
```

IX.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. .

```
displayGreeting() {
	print("Gooding Morning")
}

int add(int n1, int n2) {
	return n1 + n2
}

//Still works if you omit the types:

add(int n1, int n2) {
	return n1 + n2
}
```

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

```
int add(int n1, int n2) => n1 + n2 ;
```

Note

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

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.

A function can have two types of parameters: **required and optional.** The required parameters are listed first, followed by any optional parameters. Optional parameters can be named or positional, but not both.

Use { } to indicate the parameter are optional

Eg. Using Named parameter

```
int add(int n1, int n2, {int multiplier }) {
      return  multiplier == null ?  n1 + n2 : ( n1 + n2 ) * multiplier;
}

var r1 = add ( 1, 2 , multiplier: 5 ) ;
```

Although named parameters are a kind of optional parameter, you can annotate them with @required to indicate that the parameter is mandatory — that users must provide a value for the parameter. For example:

```
int add2(int n1, int n2, {int multiplier, @required int deduction }) {

      return  multiplier == null ?  n1 + n2 : ( n1 + n2 ) * multiplier -
deduction ;

}


var r1 = add2 ( 7, 2 , deduction: 1  ) ;
```

Eg. Using Positional parameter

Wrapping a set of function parameters in [] marks them as optional positional parameters:

Eg.

```
int add3(int n1, int n2, [int multiplier]) {
     return   multiplier == null ?  n1+ n2 : ( n1 + n2 ) * multiplier;
 }
  print(add3(1,7,8));    //Noted that no named is required
```

Eg.

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
```

```
    result = '$result with a $device';
  }
  return result;
}
```

Note: Default parameter values

Your function can use = to define default values for both named and positional parameters. The default values must be compile-time constants. If no default value is provided, the default value is `null`.

X. Special Function Types

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.

Eg

```
void main() {

}
```

Eg.

```
void main(List<String> arguments) {

      print(arguments);

}
```

Functions as first-class objects

You can pass a function as a parameter to another function.

Eg

```
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()} !!!';

 loudify('hello')
```

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. The function, invoked for each item in the list, prints a string that includes the value at the specified index.

```
var list = ['apples', 'bananas', 'oranges'];
```

```
list.forEach(   (item) {

   print('${list.indexOf(item)}: $item');

}  );
```

//If the function contains only one statement, you can shorten it using arrow notation.

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

//Same as above result
```

Lexical scope & Lexical closures

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(num addBy) {
  return (num 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() {
  var 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);
}
```

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. In general, you should use it as a shorthand for an `is` test on an object following by an expression using that object.

XI.Type Checking and Casting

Eg,

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

You can make the code shorter using the `as` operator:

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

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

XII.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 ?:.

```
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';
  }
}
```

XIII.Cascade notation (..)

Cascades (..) allow you to make a sequence of operations on the same object. In addition to function calls, you can also access fields 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:

```
querySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
```

```
  ..onClick.listen((e) => window.alert('Confirmed!'));
```
The first method call, querySelector(), returns a selector object. The code that follows the cascade notation operates on this selector object, ignoring any subsequent values that might be returned.

The previous example is equivalent to:

```
var button = querySelector('#confirm');
button.text = 'Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));
```
You can also nest your 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 is not an operator. It's just part of the Dart syntax.

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();
}
```

Unlike JavaScript, conditions must use 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));
}
callbacks.forEach((c) => 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, you can use the forEach() method. Using `forEach()` is a good option if you don't need to know the current iteration counter:

```
candidates.forEach((candidate) => candidate.interview());
```
Iterable classes such as List and Set also support the `for-in` form of iteration:

```
var collection = [0, 1, 2];
for (var x in collection) {
  print(x); // 0 1 2
}
```

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.

Note: Switch statements in Dart are intended for limited circumstances, such as in interpreters or scanners.
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`.

```
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 dartdevc typically enable assertions by default.
- Some tools, such as dart and dart2js, 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 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.

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 descend from Object. *Mixin-based inheritance* means that although every class (except for Object) has exactly one superclass, a class body can be reused in multiple class hierarchies.

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);

// Set the value of the instance variable y.
p.y = 3;

// Get the value of y.
assert(p.y == 3);

// Invoke distanceTo() on p.
num 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 its y value to 4.
p?.y = 4;
```

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});
```

Version note: The `new` keyword became optional in Dart 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!
```

Version note: The `const` keyword became optional within a constant context in Dart 2.

Getting an object's type

To get an object's type at runtime, you can use Object's `runtimeType` property, which returns a Type object.

```
print('The type of a is ${a.runtimeType}');
```
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 {
  num x; // Declare instance variable x, initially null.
  num y; // Declare y, initially null.
  num 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 also generate an implicit *setter* method. For details, see Getters and setters.

```
class Point {
  num x;
  num y;
}

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.
}
```
If you initialize an instance variable where it is declared (instead of in a constructor or method), the value is set when the instance is created, which is before the constructor and its initializer list execute.

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 {
  num x, y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    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`.
The pattern of assigning a constructor argument to an instance variable is so common, Dart has syntactic sugar to make it easy:

```
class Point {
  num x, y;

  // Syntactic sugar for setting x and y
  // before the constructor body runs.
  Point(this.x, this.y);
}
```
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:

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```
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.

Dart Tests

Hint Format Reset play_arrow Run

- check Show tests

No Yes

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(getDefaultData());
  // ...
}
```

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

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, num> json)
    : x = json['x'],
      y = json['y'] {
  print('In Point.fromJson(): ($x, $y)');
}
```

Warning: The right-hand side of an initializer does not 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.

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 appearing after a colon (:).

```
class Point {
  num x, y;

  // The main constructor for this class.
  Point(this.x, this.y);

  // Delegates to the main constructor.
  Point.alongXAxis(num 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 final ImmutablePoint origin =
      const ImmutablePoint(0, 0);

  final num 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.

The following example demonstrates a factory constructor returning objects from a cache:

```
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));
  }

  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');
```

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 {
  num x, y;

  Point(this.x, this.y);

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

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 {
  num left, top, width, height;

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

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

Overriding members

Subclasses can override instance methods, getters, and setters. You can use the `@override` annotation to indicate that you are intentionally overriding a member:

```
class SmartTelevision extends Television {
  @override
  void turnOn() {...}
  // ···
}
```

To narrow the type of a method parameter or instance variable in code that is type safe, you can use the `covariant` keyword.

Overridable operators

You can override the operators shown in the following table. For example, if you define a Vector class, you might define a `+` method to add two vectors.

`<`	`+`	`|`	`[]`
`>`	`/`	`^`	`[]=`
`<=`	`~/`	`&`	`~`
`>=`	`*`	`<<`	
`–`	`%`	`>>`	

Note: You may have noticed that `!=` is not an overridable operator. The expression `e1 != e2` is just syntactic sugar for `!(e1 == e2)`.

Here's an example of a class that overrides the `+` and `–` operators:

```
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);

  // Operator == and hashCode not shown. For details, see note
below.
  // ···
}

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

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

If you override `==`, you should also override Object's `hashCode` getter. For an example of overriding `==` and `hashCode`, see Implementing map keys.

For more information on overriding, in general, see Extending a class.

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 implemention of `noSuchMethod()` that's different from the one in class `Object`.

For more information, see the informal noSuchMethod forwarding specification.

Enumerated types

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

Using enums

Declare an enumerated type using the `enum` keyword:

```
enum Color { red, green, 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 of the values in the enum, 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'
}
```

Enumerated types have the following limits:

- You can't subclass, mix in, or implement an enum.
- You can't explicitly instantiate an enum.

For more information, see the Dart language specification.

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');
    }
  }
}
```

To specify that only certain types can use the mixin — for example, so your mixin can invoke a method that it doesn't define — use `on` to specify the required superclass:

```
mixin MusicalPerformer on Musician {
  // ...
}
```

Version note: Support for the `mixin` keyword was introduced in Dart 2.1. Code in earlier releases usually used `abstract class` instead. For more information on 2.1 mixin changes, see the Dart SDK changelog and 2.1 mixin specification.

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) do not operate on an instance, and thus do not have access to `this`. For example:

```
import 'dart:math';

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

  static num 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:

```
var names = List<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 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 = List<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 of its parameters. You can do this using `extends`.

```
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 subclasses as 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:

```
var foo = Foo<Object>();
```

Using generic methods

Initially, Dart's generic support was limited to classes. A newer syntax, called *generic methods*, allows type arguments on methods and functions:

```
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`).

For more information about generics, see Using Generic Methods.

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.

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';
```

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 dart2js supports deferred loading. Flutter, the Dart VM, and dartdevc don't support deferred loading. For more information, 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 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.

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.
- When to use the `library` directive.

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`.
- 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 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 (details). 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`:

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

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`:

```
Future main() async {
  // ...
  await for (var 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.

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.

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 isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate.

For more information, see the following:

- Dart asynchronous programming: Isolates and event loops
- dart:isolate API reference, including Isolate.spawn() and TransferableTypedData
- Background parsing cookbook on the Flutter site
- Isolate sample app

Typedefs

In Dart, functions are objects, just like strings and numbers are objects. A *typedef*, or *function-type alias*, gives a function type a name that you can use when declaring fields and return types. A typedef retains type information when a function type is assigned to a variable.

Consider the following code, which doesn't use a typedef:

```
class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    compare = f;
  }
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);

  // All we know is that compare is a function,
  // but what type of function?
  assert(coll.compare is Function);
}
```

Type information is lost when assigning `f` to `compare`. The type of `f` is `(Object, Object) → int` (where → means returns), yet the type of `compare` is Function. If we change the code to use explicit names and retain type information, both developers and tools can use that information.

```
typedef Compare = int Function(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);
  assert(coll.compare is Function);
  assert(coll.compare is Compare);
}
```

Note: Currently, typedefs are restricted to function types. We expect this to change.

Because typedefs are simply aliases, they offer a way to check the type of any function. For example:

```
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.

Two annotations are available to all Dart code: `@deprecated` and `@override`. For examples of using `@override`, see Extending a class. Here's an example of using the `@deprecated` annotation:

```
class Television {
  /// _Deprecated: Use [turnOn] instead._
  @deprecated
  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:

```
library todo;

class Todo {
  final String who;
  final String what;

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

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

```
import 'todo.dart';

@Todo('seth', 'make this do something')
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 Dart compiler 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.
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 generated documentation, `[Food]` becomes a link to the API docs for the Food class.

To parse Dart code and generate HTML documentation, you can use the SDK's documentation generation tool. For an example of generated documentation, see the Dart API documentation. For advice on how to structure your comments, see Guidelines for Dart Doc Comments.

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.

A tour of the core libraries

This page shows you how to use the major features in Dart's core libraries. It's just an overview, and by no means comprehensive. Whenever you need more details about a class, consult the Dart API reference.

dart:core
Built-in types, collections, and other core functionality. This library is automatically imported into every Dart program.
dart:async
Support for asynchronous programming, with classes such as Future and Stream.
dart:math
Mathematical constants and functions, plus a random number generator.
dart:convert
Encoders and decoders for converting between different data representations, including JSON and UTF-8.
dart:html
DOM and other APIs for browser-based apps.
dart:io
I/O for programs that can use the Dart VM, including Flutter apps, servers, and command-line scripts.
This page is just an overview; it covers only a few dart:* libraries and no third-party libraries.

Other places to find library information are the pub.dev site and the Dart web developer library guide. You can find API documentation for all dart:* libraries in the Dart API reference or, if you're using Flutter, the Flutter API reference.

DartPad tip: You can play with the code in this page by copying it into a DartPad.

dart:core - numbers, collections, strings, and more

The dart:core library (API reference) provides a small but critical set of built-in functionality. This library is automatically imported into every Dart program.

Printing to the console

The top-level `print()` method takes a single argument (any Object) and displays that object's string value (as returned by `toString()`) in the console.

```
print(anObject);
print('I drink $tea.');
```

For more information on basic strings and `toString()`, see Strings in the language tour.

Numbers

The dart:core library defines the num, int, and double classes, which have some basic utilities for working with numbers.

You can convert a string into an integer or double with the `parse()` methods of int and double, respectively:

```
assert(int.parse('42') == 42);
assert(int.parse('0x42') == 66);
assert(double.parse('0.50') == 0.5);
```

Or use the parse() method of num, which creates an integer if possible and otherwise a double:

```
assert(num.parse('42') is int);
assert(num.parse('0x42') is int);
assert(num.parse('0.50') is double);
```

To specify the base of an integer, add a `radix` parameter:

```
assert(int.parse('42', radix: 16) == 66);
```

Use the `toString()` method to convert an int or double to a string. To specify the number of digits to the right of the decimal, use toStringAsFixed(). To specify the number of significant digits in the string, use toStringAsPrecision():

```
// Convert an int to a string.
```

```
assert(42.toString() == '42');

// Convert a double to a string.
assert(123.456.toString() == '123.456');

// Specify the number of digits after the decimal.
assert(123.456.toStringAsFixed(2) == '123.46');

// Specify the number of significant figures.
assert(123.456.toStringAsPrecision(2) == '1.2e+2');
assert(double.parse('1.2e+2') == 120.0);
```

For more information, see the API documentation for int, double, and num. Also see the dart:math section.

Strings and regular expressions

A string in Dart is an immutable sequence of UTF-16 code units. The language tour has more information about strings. You can use regular expressions (RegExp objects) to search within strings and to replace parts of strings.

The String class defines such methods as `split()`, `contains()`, `startsWith()`, `endsWith()`, and more.

Searching inside a string

You can find particular locations within a string, as well as check whether a string begins with or ends with a particular pattern. For example:

```
// Check whether a string contains another string.
assert('Never odd or even'.contains('odd'));

// Does a string start with another string?
assert('Never odd or even'.startsWith('Never'));

// Does a string end with another string?
assert('Never odd or even'.endsWith('even'));

// Find the location of a string inside a string.
assert('Never odd or even'.indexOf('odd') == 6);
```

Extracting data from a string

You can get the individual characters from a string as Strings or ints, respectively. To be precise, you actually get individual UTF-16 code units; high-numbered characters such as the treble clef symbol ('\u{1D11E}') are two code units apiece.

You can also extract a substring or split a string into a list of substrings:

```
// Grab a substring.
assert('Never odd or even'.substring(6, 9) == 'odd');

// Split a string using a string pattern.
var parts = 'structured web apps'.split(' ');
assert(parts.length == 3);
assert(parts[0] == 'structured');

// Get a UTF-16 code unit (as a string) by index.
assert('Never odd or even'[0] == 'N');

// Use split() with an empty string parameter to get
// a list of all characters (as Strings); good for
// iterating.
for (var char in 'hello'.split('')) {
  print(char);
}

// Get all the UTF-16 code units in the string.
var codeUnitList =
    'Never odd or even'.codeUnits.toList();
assert(codeUnitList[0] == 78);
```

Converting to uppercase or lowercase

You can easily convert strings to their uppercase and lowercase variants:

```
// Convert to uppercase.
assert('structured web apps'.toUpperCase() ==
    'STRUCTURED WEB APPS');

// Convert to lowercase.
assert('STRUCTURED WEB APPS'.toLowerCase() ==
    'structured web apps');
```

Note: These methods don't work for every language. For example, the Turkish alphabet's dotless *ı* is converted incorrectly.

Trimming and empty strings

Remove all leading and trailing white space with `trim()`. To check whether a string is empty (length is zero), use `isEmpty`.

```
// Trim a string.
assert('  hello  '.trim() == 'hello');

// Check whether a string is empty.
assert(''.isEmpty);

// Strings with only white space are not empty.
assert('  '.isNotEmpty);
```

Replacing part of a string

Strings are immutable objects, which means you can create them but you can't change them. If you look closely at the String API reference, you'll notice that none of the methods actually changes the state of a String. For example, the method `replaceAll()` returns a new String without changing the original String:

```
var greetingTemplate = 'Hello, NAME!';
var greeting =
    greetingTemplate.replaceAll(RegExp('NAME'), 'Bob');

// greetingTemplate didn't change.
assert(greeting != greetingTemplate);
```

Building a string

To programmatically generate a string, you can use StringBuffer. A StringBuffer doesn't generate a new String object until `toString()` is called. The `writeAll()` method has an optional second parameter that lets you specify a separator—in this case, a space.

```
var sb = StringBuffer();
sb
  ..write('Use a StringBuffer for ')
  ..writeAll(['efficient', 'string', 'creation'], ' ')
  ..write('.');

var fullString = sb.toString();

assert(fullString ==
    'Use a StringBuffer for efficient string creation.');
```

Regular expressions

The RegExp class provides the same capabilities as JavaScript regular expressions. Use regular expressions for efficient searching and pattern matching of strings.

```
// Here's a regular expression for one or more digits.
var numbers = RegExp(r'\d+');

var allCharacters = 'llamas live fifteen to twenty years';
var someDigits = 'llamas live 15 to 20 years';

// contains() can use a regular expression.
assert(!allCharacters.contains(numbers));
assert(someDigits.contains(numbers));

// Replace every match with another string.
var exedOut = someDigits.replaceAll(numbers, 'XX');
assert(exedOut == 'llamas live XX to XX years');
```

You can work directly with the RegExp class, too. The Match class provides access to a regular expression match.

```
var numbers = RegExp(r'\d+');
var someDigits = 'llamas live 15 to 20 years';

// Check whether the reg exp has a match in a string.
assert(numbers.hasMatch(someDigits));

// Loop through all matches.
for (var match in numbers.allMatches(someDigits)) {
  print(match.group(0)); // 15, then 20
}
```

More information

Refer to the String API reference for a full list of methods. Also see the API reference for StringBuffer, Pattern, RegExp, and Match.

Collections

Dart ships with a core collections API, which includes classes for lists, sets, and maps.

Lists

As the language tour shows, you can use literals to create and initialize lists. Alternatively, use one of the List constructors. The List class also defines several methods for adding items to and removing items from lists.

```
// Use a List constructor.
var vegetables = List();

// Or simply use a list literal.
var fruits = ['apples', 'oranges'];

// Add to a list.
fruits.add('kiwis');

// Add multiple items to a list.
fruits.addAll(['grapes', 'bananas']);

// Get the list length.
assert(fruits.length == 5);

// Remove a single item.
var appleIndex = fruits.indexOf('apples');
fruits.removeAt(appleIndex);
assert(fruits.length == 4);
```

```
// Remove all elements from a list.
fruits.clear();
assert(fruits.isEmpty);
```
Use `indexOf()` to find the index of an object in a list:

```
var fruits = ['apples', 'oranges'];

// Access a list item by index.
assert(fruits[0] == 'apples');

// Find an item in a list.
assert(fruits.indexOf('apples') == 0);
```
Sort a list using the `sort()` method. You can provide a sorting function that compares two objects. This sorting function must return < 0 for *smaller*, 0 for the *same*, and > 0 for *bigger*. The following example uses `compareTo()`, which is defined by Comparable and implemented by String.

```
var fruits = ['bananas', 'apples', 'oranges'];

// Sort a list.
fruits.sort((a, b) => a.compareTo(b));
assert(fruits[0] == 'apples');
```
Lists are parameterized types, so you can specify the type that a list should contain:

```
// This list should contain only strings.
var fruits = List<String>();

fruits.add('apples');
var fruit = fruits[0];
assert(fruit is String);
fruits.add(5); // Error: 'int' can't be assigned to 'String'
```
Refer to the List API reference for a full list of methods.

Sets

A set in Dart is an unordered collection of unique items. Because a set is unordered, you can't get a set's items by index (position).

```
var ingredients = Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
assert(ingredients.length == 3);

// Adding a duplicate item has no effect.
ingredients.add('gold');
assert(ingredients.length == 3);

// Remove an item from a set.
```

```
ingredients.remove('gold');
assert(ingredients.length == 2);
```
Use `contains()` and `containsAll()` to check whether one or more objects are in a set:

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

// Check whether an item is in the set.
assert(ingredients.contains('titanium'));

// Check whether all the items are in the set.
assert(ingredients.containsAll(['titanium', 'xenon']));
```
An intersection is a set whose items are in two other sets.

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

// Create the intersection of two sets.
var nobleGases = Set.from(['xenon', 'argon']);
var intersection = ingredients.intersection(nobleGases);
assert(intersection.length == 1);
assert(intersection.contains('xenon'));
```
Refer to the Set API reference for a full list of methods.

Maps

A map, commonly known as a *dictionary* or *hash*, is an unordered collection of key-value pairs. Maps associate a key to some value for easy retrieval. Unlike in JavaScript, Dart objects are not maps.

You can declare a map using a terse literal syntax, or you can use a traditional constructor:

```
// Maps often use strings as keys.
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

// Maps can be built from a constructor.
var searchTerms = Map();

// Maps are parameterized types; you can specify what
// types the key and value should be.
var nobleGases = Map<int, String>();
```
You add, get, and set map items using the bracket syntax.
Use `remove()` to remove a key and its value from a map.

```
var nobleGases = {54: 'xenon'};

// Retrieve a value with a key.
assert(nobleGases[54] == 'xenon');

// Check whether a map contains a key.
assert(nobleGases.containsKey(54));

// Remove a key and its value.
nobleGases.remove(54);
assert(!nobleGases.containsKey(54));
```

You can retrieve all the values or all the keys from a map:

```
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

// Get all the keys as an unordered collection
// (an Iterable).
var keys = hawaiianBeaches.keys;

assert(keys.length == 3);
assert(Set.from(keys).contains('Oahu'));

// Get all the values as an unordered collection
// (an Iterable of Lists).
var values = hawaiianBeaches.values;
assert(values.length == 3);
assert(values.any((v) => v.contains('Waikiki')));
```

To check whether a map contains a key, use `containsKey()`. Because map values can be null, you cannot rely on simply getting the value for the key and checking for null to determine the existence of a key.

```
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

assert(hawaiianBeaches.containsKey('Oahu'));
assert(!hawaiianBeaches.containsKey('Florida'));
```

Use the `putIfAbsent()` method when you want to assign a value to a key if and only if the key does not already exist in a map. You must provide a function that returns the value.

```
var teamAssignments = {};
teamAssignments.putIfAbsent(
    'Catcher', () => pickToughestKid());
```

```
assert(teamAssignments['Catcher'] != null);
```
Refer to the Map API reference for a full list of methods.

Common collection methods

List, Set, and Map share common functionality found in many collections. Some of this common functionality is defined by the Iterable class, which List and Set implement.

Note: Although Map doesn't implement Iterable, you can get Iterables from it using the Map `keys` and `values` properties.

Use `isEmpty` or `isNotEmpty` to check whether a list, set, or map has items:

```
var coffees = [];
var teas = ['green', 'black', 'chamomile', 'earl grey'];
assert(coffees.isEmpty);
assert(teas.isNotEmpty);
```
To apply a function to each item in a list, set, or map, you can use `forEach()`:

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

teas.forEach((tea) => print('I drink $tea'));
```
When you invoke `forEach()` on a map, your function must take two arguments (the key and value):

```
hawaiianBeaches.forEach((k, v) {
  print('I want to visit $k and swim at $v');
  // I want to visit Oahu and swim at
  // [Waikiki, Kailua, Waimanalo], etc.
});
```
Iterables provide the `map()` method, which gives you all the results in a single object:

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

var loudTeas = teas.map((tea) => tea.toUpperCase());
loudTeas.forEach(print);
```
Note: The object returned by `map()` is an Iterable that's *lazily evaluated*: your function isn't called until you ask for an item from the returned object.

To force your function to be called immediately on each item, use `map().toList()` or `map().toSet()`:

```
var loudTeas =
    teas.map((tea) => tea.toUpperCase()).toList();
```

Use Iterable's `where()` method to get all the items that match a condition. Use Iterable's `any()` and `every()` methods to check whether some or all items match a condition.

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

// Chamomile is not caffeinated.
bool isDecaffeinated(String teaName) =>
    teaName == 'chamomile';

// Use where() to find only the items that return true
// from the provided function.
var decaffeinatedTeas =
    teas.where((tea) => isDecaffeinated(tea));
// or teas.where(isDecaffeinated)

// Use any() to check whether at least one item in the
// collection satisfies a condition.
assert(teas.any(isDecaffeinated));

// Use every() to check whether all the items in a
// collection satisfy a condition.
assert(!teas.every(isDecaffeinated));
```

For a full list of methods, refer to the Iterable API reference, as well as those for List, Set, and Map.

URIs

The Uri class provides functions to encode and decode strings for use in URIs (which you might know as *URLs*). These functions handle characters that are special for URIs, such as `&` and `=`. The Uri class also parses and exposes the components of a URI—host, port, scheme, and so on.

Encoding and decoding fully qualified URIs

To encode and decode characters *except* those with special meaning in a URI (such as `/`, `:`, `&`, `#`), use the `encodeFull()` and `decodeFull()` methods. These methods are good for encoding or decoding a fully qualified URI, leaving intact special URI characters.

```
var uri = 'http://example.org/api?foo=some message';

var encoded = Uri.encodeFull(uri);
assert(encoded ==
    'http://example.org/api?foo=some%20message');

var decoded = Uri.decodeFull(encoded);
```

```
assert(uri == decoded);
```
Notice how only the space between `some` and `message` was encoded.

Encoding and decoding URI components

To encode and decode all of a string's characters that have special meaning in a URI, including (but not limited to) `/`, `&`, and `:`, use the `encodeComponent()` and `decodeComponent()` methods.

```
var uri = 'http://example.org/api?foo=some message';

var encoded = Uri.encodeComponent(uri);
assert(encoded ==
    'http%3A%2F%2Fexample.org%2Fapi%3Ffoo%3Dsome%20message');

var decoded = Uri.decodeComponent(encoded);
assert(uri == decoded);
```
Notice how every special character is encoded. For example, `/` is encoded to `%2F`.

Parsing URIs

If you have a Uri object or a URI string, you can get its parts using Uri fields such as `path`. To create a Uri from a string, use the `parse()` static method:

```
var uri =
    Uri.parse('http://example.org:8080/foo/bar#frag');

assert(uri.scheme == 'http');
assert(uri.host == 'example.org');
assert(uri.path == '/foo/bar');
assert(uri.fragment == 'frag');
assert(uri.origin == 'http://example.org:8080');
```
See the Uri API reference for more URI components that you can get.

Building URIs

You can build up a URI from individual parts using the `Uri()` constructor:

```
var uri = Uri(
    scheme: 'http',
    host: 'example.org',
    path: '/foo/bar',
    fragment: 'frag');
assert(
    uri.toString() == 'http://example.org/foo/bar#frag');
```

Dates and times

A DateTime object is a point in time. The time zone is either UTC or the local time zone.

You can create DateTime objects using several constructors:

```
// Get the current date and time.
var now = DateTime.now();

// Create a new DateTime with the local time zone.
var y2k = DateTime(2000); // January 1, 2000

// Specify the month and day.
y2k = DateTime(2000, 1, 2); // January 2, 2000

// Specify the date as a UTC time.
y2k = DateTime.utc(2000); // 1/1/2000, UTC

// Specify a date and time in ms since the Unix epoch.
y2k = DateTime.fromMillisecondsSinceEpoch(946684800000,
    isUtc: true);

// Parse an ISO 8601 date.
y2k = DateTime.parse('2000-01-01T00:00:00Z');
```

The `millisecondsSinceEpoch` property of a date returns the number of milliseconds since the “Unix epoch”—January 1, 1970, UTC:

```
// 1/1/2000, UTC
var y2k = DateTime.utc(2000);
assert(y2k.millisecondsSinceEpoch == 946684800000);

// 1/1/1970, UTC
var unixEpoch = DateTime.utc(1970);
assert(unixEpoch.millisecondsSinceEpoch == 0);
```

Use the Duration class to calculate the difference between two dates and to shift a date forward or backward:

```
var y2k = DateTime.utc(2000);

// Add one year.
var y2001 = y2k.add(Duration(days: 366));
assert(y2001.year == 2001);

// Subtract 30 days.
var december2000 = y2001.subtract(Duration(days: 30));
assert(december2000.year == 2000);
assert(december2000.month == 12);

// Calculate the difference between two dates.
// Returns a Duration object.
var duration = y2001.difference(y2k);
```

```
assert(duration.inDays == 366); // y2k was a leap year.
```
Warning: Using a Duration to shift a DateTime by days can be problematic, due to clock shifts (to daylight saving time, for example). Use UTC dates if you must shift days.
For a full list of methods, refer to the API reference for DateTime and Duration.

Utility classes

The core library contains various utility classes, useful for sorting, mapping values, and iterating.

Comparing objects

Implement the Comparable interface to indicate that an object can be compared to another object, usually for sorting.
The `compareTo()` method returns < 0 for *smaller*, 0 for the *same*, and > 0 for *bigger*.

```
class Line implements Comparable<Line> {
  final int length;
  const Line(this.length);

  @override
  int compareTo(Line other) => length - other.length;
}

void main() {
  var short = const Line(1);
  var long = const Line(100);
  assert(short.compareTo(long) < 0);
}
```

Implementing map keys

Each object in Dart automatically provides an integer hash code, and thus can be used as a key in a map. However, you can override the `hashCode` getter to generate a custom hash code. If you do, you might also want to override the `==` operator. Objects that are equal (via `==`) must have identical hash codes. A hash code doesn't have to be unique, but it should be well distributed.

```
class Person {
  final String firstName, lastName;

  Person(this.firstName, this.lastName);

  // Override hashCode using strategy from Effective Java,
  // Chapter 11.
```

```
  @override
  int get hashCode {
    int result = 17;
    result = 37 * result + firstName.hashCode;
    result = 37 * result + lastName.hashCode;
    return result;
  }

  // You should generally implement operator == if you
  // override hashCode.
  @override
  bool operator ==(dynamic other) {
    if (other is! Person) return false;
    Person person = other;
    return (person.firstName == firstName &&
        person.lastName == lastName);
  }
}

void main() {
  var p1 = Person('Bob', 'Smith');
  var p2 = Person('Bob', 'Smith');
  var p3 = 'not a person';
  assert(p1.hashCode == p2.hashCode);
  assert(p1 == p2);
  assert(p1 != p3);
}
```

Iteration

The Iterable and Iterator classes support for-in loops. Extend (if possible) or implement Iterable whenever you create a class that can provide Iterators for use in for-in loops. Implement Iterator to define the actual iteration ability.

```
class Process {
  // Represents a process...
}

class ProcessIterator implements Iterator<Process> {
  @override
  Process get current => ...
  @override
  bool moveNext() => ...
}

// A mythical class that lets you iterate through all
// processes. Extends a subclass of [Iterable].
class Processes extends IterableBase<Process> {
  @override
  final Iterator<Process> iterator = ProcessIterator();
}
```

```
void main() {
  // Iterable objects can be used with for-in.
  for (var process in Processes()) {
    // Do something with the process.
  }
}
```

Exceptions

The Dart core library defines many common exceptions and errors. Exceptions are considered conditions that you can plan ahead for and catch. Errors are conditions that you don't expect or plan for.

A couple of the most common errors are:

NoSuchMethodError
Thrown when a receiving object (which might be null) does not implement a method.

ArgumentError
Can be thrown by a method that encounters an unexpected argument.

Throwing an application-specific exception is a common way to indicate that an error has occurred. You can define a custom exception by implementing the Exception interface:

```
class FooException implements Exception {
  final String msg;

  const FooException([this.msg]);

  @override
  String toString() => msg ?? 'FooException';
}
```

For more information, see Exceptions (in the language tour) and the Exception API reference.

dart:async - asynchronous programming

Asynchronous programming often uses callback functions, but Dart provides alternatives: Future and Stream objects. A Future is like a promise for a result to be provided sometime in the future. A Stream is a way to get a sequence of values, such as events. Future, Stream, and more are in the dart:async library (API reference).

Note: You don't always need to use the Future or Stream APIs directly. The Dart language supports asynchronous coding using keywords such as `async` and `await`. See the asynchronous programming codelab for details.
The dart:async library works in both web apps and command-line apps. To use it, import dart:async:

```
import 'dart:async';
```
Version note: As of Dart 2.1, you don't need to import dart:async to use the Future and Stream APIs, because dart:core exports those classes.

Future

Future objects appear throughout the Dart libraries, often as the object returned by an asynchronous method. When a future *completes*, its value is ready to use.

Using await

Before you directly use the Future API, consider using `await` instead. Code that uses `await` expressions can be easier to understand than code that uses the Future API.

Consider the following function. It uses Future's `then()` method to execute three asynchronous functions in a row, waiting for each one to complete before executing the next one.

```
runUsingFuture() {
  // ...
  findEntryPoint().then((entryPoint) {
    return runExecutable(entryPoint, args);
  }).then(flushThenExit);
}
```
The equivalent code with await expressions looks more like synchronous code:

```
runUsingAsyncAwait() async {
  // ...
  var entryPoint = await findEntryPoint();
  var exitCode = await runExecutable(entryPoint, args);
  await flushThenExit(exitCode);
}
```
An `async` function can catch exceptions from Futures. For example:

```
var entryPoint = await findEntryPoint();
try {
  var exitCode = await runExecutable(entryPoint, args);
  await flushThenExit(exitCode);
```

```
} catch (e) {
  // Handle the error...
}
```
Important: Async functions return Futures. If you don't want your function to return a future, then use a different solution. For example, you might call an `async` function from your function.
For more information on using `await` and related Dart language features, see the asynchronous programming codelab.

Basic usage

You can use `then()` to schedule code that runs when the future completes. For example, `HttpRequest.getString()` returns a Future, since HTTP requests can take a while. Using `then()` lets you run some code when that Future has completed and the promised string value is available:

```
HttpRequest.getString(url).then((String result) {
  print(result);
});
```
Use `catchError()` to handle any errors or exceptions that a Future object might throw.

```
HttpRequest.getString(url).then((String result) {
  print(result);
}).catchError((e) {
  // Handle or ignore the error.
});
```
The `then().catchError()` pattern is the asynchronous version of `try-catch`.

Important: Be sure to invoke `catchError()` on the result of `then()`—not on the result of the original Future. Otherwise, the `catchError()` can handle errors only from the original Future's computation, but not from the handler registered by `then()`.

Chaining multiple asynchronous methods

The `then()` method returns a Future, providing a useful way to run multiple asynchronous functions in a certain order. If the callback registered with `then()` returns a Future, `then()` returns an equivalent Future. If the callback returns a value of any other type, `then()` creates a new Future that completes with the value.

```
Future result = costlyQuery(url);
result
    .then((value) => expensiveWork(value))
    .then((_) => lengthyComputation())
    .then((_) => print('Done!'))
```

```
    .catchError((exception) {
  /* Handle exception... */
});
```
In the preceding example, the methods run in the following order:

1. `costlyQuery()`
2. `expensiveWork()`
3. `lengthyComputation()`

Here is the same code written using await:

```
try {
  final value = await costlyQuery(url);
  await expensiveWork(value);
  await lengthyComputation();
  print('Done!');
} catch (e) {
  /* Handle exception... */
}
```
Waiting for multiple futures

Sometimes your algorithm needs to invoke many asynchronous functions and wait for them all to complete before continuing. Use the Future.wait() static method to manage multiple Futures and wait for them to complete:

```
Future deleteLotsOfFiles() async =>  ...
Future copyLotsOfFiles() async =>  ...
Future checksumLotsOfOtherFiles() async =>  ...

await Future.wait([
  deleteLotsOfFiles(),
  copyLotsOfFiles(),
  checksumLotsOfOtherFiles(),
]);
print('Done with all the long steps!');
```
Stream

Stream objects appear throughout Dart APIs, representing sequences of data. For example, HTML events such as button clicks are delivered using streams. You can also read a file as a stream.

Using an asynchronous for loop

Sometimes you can use an asynchronous for loop (`await for`) instead of using the Stream API.

Consider the following function. It uses Stream's `listen()` method to subscribe to a list of files, passing in a function literal that searches each file or directory.

```
void main(List<String> arguments) {
  // ...
  FileSystemEntity.isDirectory(searchPath).then((isDir) {
    if (isDir) {
      final startingDir = Directory(searchPath);
      startingDir
          .list(
              recursive: argResults[recursive],
              followLinks: argResults[followLinks])
          .listen((entity) {
        if (entity is File) {
          searchFile(entity, searchTerms);
        }
      });
    } else {
      searchFile(File(searchPath), searchTerms);
    }
  });
}
```

The equivalent code with await expressions, including an asynchronous for loop (`await for`), looks more like synchronous code:

```
Future main(List<String> arguments) async {
  // ...
  if (await FileSystemEntity.isDirectory(searchPath)) {
    final startingDir = Directory(searchPath);
    await for (var entity in startingDir.list(
        recursive: argResults[recursive],
        followLinks: argResults[followLinks])) {
      if (entity is File) {
        searchFile(entity, searchTerms);
      }
    }
  } else {
    searchFile(File(searchPath), searchTerms);
  }
}
```

Important: Before using `await for`, make 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 DOM event listeners, because the DOM sends endless streams of events. If you use `await for` to register two DOM event listeners in a row, then the second kind of event is never handled.

For more information on using `await` and related Dart language features, see the asynchronous programming codelab.

Listening for stream data

To get each value as it arrives, either use `await for` or subscribe to the stream using the `listen()` method:

```
// Find a button by ID and add an event handler.
querySelector('#submitInfo').onClick.listen((e) {
  // When the button is clicked, it runs this code.
  submitData();
});
```

In this example, the `onClick` property is a Stream object provided by the "submitInfo" button.

If you care about only one event, you can get it using a property such as `first`, `last`, or `single`. To test the event before handling it, use a method such as `firstWhere()`, `lastWhere()`, or `singleWhere()`.

If you care about a subset of events, you can use methods such as `skip()`, `skipWhile()`, `take()`, `takeWhile()`, and `where()`.

Transforming stream data

Often, you need to change the format of a stream's data before you can use it. Use the `transform()` method to produce a stream with a different type of data:

```
var lines = inputStream
    .transform(utf8.decoder)
    .transform(LineSplitter());
```

This example uses two transformers. First it uses utf8.decoder to transform the stream of integers into a stream of strings. Then it uses a LineSplitter to transform the stream of strings into a stream of separate lines. These transformers are from the dart:convert library (see the dart:convert section).

Handling errors and completion

How you specify error and completion handling code depends on whether you use an asynchronous for loop (`await for`) or the Stream API.

If you use an asynchronous for loop, then use try-catch to handle errors. Code that executes after the stream is closed goes after the asynchronous for loop.

```
Future readFileAwaitFor() async {
  var config = File('config.txt');
  Stream<List<int>> inputStream = config.openRead();

  var lines = inputStream
```

```
      .transform(utf8.decoder)
      .transform(LineSplitter());
  try {
    await for (var line in lines) {
      print('Got ${line.length} characters from stream');
    }
    print('file is now closed');
  } catch (e) {
    print(e);
  }
}
```

If you use the Stream API, then handle errors by registering an `onError` listener. Run code after the stream is closed by registering an `onDone` listener.

```
var config = File('config.txt');
Stream<List<int>> inputStream = config.openRead();

inputStream
    .transform(utf8.decoder)
    .transform(LineSplitter())
    .listen((String line) {
  print('Got ${line.length} characters from stream');
}, onDone: () {
  print('file is now closed');
}, onError: (e) {
  print(e);
});
```

More information

For some examples of using Future and Stream in command-line apps, see the dart:io tour. Also see these articles, codelabs, and tutorials:

- Asynchronous programming: futures, async, await
- Futures and error handling
- Asynchronous programming: streams
- Creating streams in Dart
- Dart asynchronous programming: Isolates and event loops

dart:math - math and random

The dart:math library (API reference) provides common functionality such as sine and cosine, maximum and minimum, and constants such as *pi* and *e*. Most of the functionality in the Math library is implemented as top-level functions.

To use this library in your app, import dart:math.

```
import 'dart:math';
```

Trigonometry

The Math library provides basic trigonometric functions:

```
// Cosine
assert(cos(pi) == -1.0);

// Sine
var degrees = 30;
var radians = degrees * (pi / 180);
// radians is now 0.52359.
var sinOf30degrees = sin(radians);
// sin 30° = 0.5
assert((sinOf30degrees - 0.5).abs() < 0.01);
```

Note: These functions use radians, not degrees!

Maximum and minimum

The Math library provides `max()` and `min()` methods:

```
assert(max(1, 1000) == 1000);
assert(min(1, -1000) == -1000);
```

Math constants

Find your favorite constants—*pi*, *e*, and more—in the Math library:

```
// See the Math library for additional constants.
print(e); // 2.718281828459045
print(pi); // 3.141592653589793
print(sqrt2); // 1.4142135623730951
```

Random numbers

Generate random numbers with the Random class. You can optionally provide a seed to the Random constructor.

```
var random = Random();
random.nextDouble(); // Between 0.0 and 1.0: [0, 1)
random.nextInt(10); // Between 0 and 9.
```

You can even generate random booleans:

```
var random = Random();
random.nextBool(); // true or false
```

More information

Refer to the Math API reference for a full list of methods. Also see the API reference for num, int, and double.

dart:convert - decoding and encoding JSON, UTF-8, and more

The dart:convert library (API reference) has converters for JSON and UTF-8, as well as support for creating additional converters. JSON is a simple text format for representing structured objects and collections. UTF-8 is a common variable-width encoding that can represent every character in the Unicode character set.

The dart:convert library works in both web apps and command-line apps. To use it, import dart:convert.

```
import 'dart:convert';
```

Decoding and encoding JSON

Decode a JSON-encoded string into a Dart object with `jsonDecode()`:

```
// NOTE: Be sure to use double quotes ("),
// not single quotes ('), inside the JSON string.
// This string is JSON, not Dart.
var jsonString = '''
  [
    {"score": 40},
    {"score": 80}
  ]
''';

var scores = jsonDecode(jsonString);
assert(scores is List);

var firstScore = scores[0];
assert(firstScore is Map);
assert(firstScore['score'] == 40);
```

Encode a supported Dart object into a JSON-formatted string with `jsonEncode()`:

```
var scores = [
  {'score': 40},
  {'score': 80},
  {'score': 100, 'overtime': true, 'special_guest': null}
```

```
];

var jsonText = jsonEncode(scores);
assert(jsonText ==
    '[{"score":40},{"score":80},'
        '{"score":100,"overtime":true,'
        '"special_guest":null}]');
```

Only objects of type int, double, String, bool, null, List, or Map (with string keys) are directly encodable into JSON. List and Map objects are encoded recursively.

You have two options for encoding objects that aren't directly encodable. The first is to invoke `encode()` with a second argument: a function that returns an object that is directly encodable. Your second option is to omit the second argument, in which case the encoder calls the object's `toJson()` method.

For more examples and links to JSON-related packages, see Using JSON.

Decoding and encoding UTF-8 characters

Use `utf8.decode()` to decode UTF8-encoded bytes to a Dart string:

```
List<int> utf8Bytes = [
  0xc3, 0x8e, 0xc3, 0xb1, 0xc5, 0xa3, 0xc3, 0xa9,
  0x72, 0xc3, 0xb1, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3,
  0xae, 0xc3, 0xb6, 0xc3, 0xb1, 0xc3, 0xa5, 0xc4,
  0xbc, 0xc3, 0xae, 0xc5, 0xbe, 0xc3, 0xa5, 0xc5,
  0xa3, 0xc3, 0xae, 0xe1, 0xbb, 0x9d, 0xc3, 0xb1
];

var funnyWord = utf8.decode(utf8Bytes);

assert(funnyWord == 'Îñţérñåţîöñåļîžåţîờñ');
```

To convert a stream of UTF-8 characters into a Dart string, specify `utf8.decoder` to the Stream `transform()` method:

```
var lines =
    utf8.decoder.bind(inputStream).transform(LineSplitter());
try {
  await for (var line in lines) {
    print('Got ${line.length} characters from stream');
  }
  print('file is now closed');
} catch (e) {
  print(e);
}
```

Use `utf8.encode()` to encode a Dart string as a list of UTF8-encoded bytes:

```
List<int> encoded = utf8.encode('Îñţérñåţîöñåļîžåţîờñ');

assert(encoded.length == utf8Bytes.length);
for (int i = 0; i < encoded.length; i++) {
  assert(encoded[i] == utf8Bytes[i]);
}
```

Other functionality

The dart:convert library also has converters for ASCII and ISO-8859-1 (Latin1). For details, see the API reference for the dart:convert library.

dart:html - browser-based apps

Use the dart:html library to program the browser, manipulate objects and elements in the DOM, and access HTML5 APIs. DOM stands for *Document Object Model*, which describes the hierarchy of an HTML page.

Other common uses of dart:html are manipulating styles (*CSS*), getting data using HTTP requests, and exchanging data using WebSockets. HTML5 (and dart:html) has many additional APIs that this section doesn't cover. Only web apps can use dart:html, not command-line apps.

Note: For a higher level approach to web app UIs, use a web framework such as AngularDart.
To use the HTML library in your web app, import dart:html:

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

Manipulating the DOM

To use the DOM, you need to know about *windows*, *documents*, *elements*, and *nodes*.

A Window object represents the actual window of the web browser. Each Window has a Document object, which points to the document that's currently loaded. The Window object also has accessors to various APIs such as IndexedDB (for storing data), requestAnimationFrame (for animations), and more. In tabbed browsers, each tab has its own Window object.

With the Document object, you can create and manipulate Element objects within the document. Note that the document itself is an element and can be manipulated.

The DOM models a tree of Nodes. These nodes are often elements, but they can also be attributes, text, comments, and other DOM types. Except for the root node, which has no parent, each node in the DOM has one parent and might have many children.

Finding elements

To manipulate an element, you first need an object that represents it. You can get this object using a query.

Find one or more elements using the top-level functions `querySelector()` and `querySelectorAll()`. You can query by ID, class, tag, name, or any combination of these. The CSS Selector Specification guide defines the formats of the selectors such as using a # prefix to specify IDs and a period (.) for classes.

The `querySelector()` function returns the first element that matches the selector, while `querySelectorAll()` returns a collection of elements that match the selector.

```
// Find an element by id (an-id).
Element elem1 = querySelector('#an-id');

// Find an element by class (a-class).
Element elem2 = querySelector('.a-class');

// Find all elements by tag (<div>).
List<Element> elems1 = querySelectorAll('div');

// Find all text inputs.
List<Element> elems2 = querySelectorAll(
  'input[type="text"]',
);

// Find all elements with the CSS class 'class'
// inside of a <p> that is inside an element with
// the ID 'id'.
List<Element> elems3 = querySelectorAll('#id p.class');
```

Manipulating elements

You can use properties to change the state of an element. Node and its subtype Element define the properties that all elements have. For example, all elements have `classes`, `hidden`, `id`, `style`, and `title` properties that you can use to set state. Subclasses of

Element define additional properties, such as the `href` property of AnchorElement.

Consider this example of specifying an anchor element in HTML:

```
<a id="example" href="http://example.com">link text</a>
```

This <a> tag specifies an element with an `href` attribute and a text node (accessible via a `text` property) that contains the string “linktext”. To change the URL that the link goes to, you can use AnchorElement's `href` property:

```
var anchor = querySelector('#example') as AnchorElement;
anchor.href = 'http://dart.dev';
```

Often you need to set properties on multiple elements. For example, the following code sets the `hidden` property of all elements that have a class of “mac”, “win”, or “linux”. Setting the `hidden` property to true has the same effect as adding `display:none` to the CSS.

```
  <!-- In HTML: -->
  <p>
    <span class="linux">Words for Linux</span>
    <span class="macos">Words for Mac</span>
    <span class="windows">Words for Windows</span>
  </p>
  // In Dart:
  final osList = ['macos', 'windows', 'linux'];
  final userOs = determineUserOs();

  // For each possible OS...
  for (var os in osList) {
    // Matches user OS?
    bool shouldShow = (os == userOs);

    // Find all elements with class=os. For example, if
    // os == 'windows', call querySelectorAll('.windows')
    // to find all elements with the class "windows".
    // Note that '.$os' uses string interpolation.
    for (var elem in querySelectorAll('.$os')) {
      elem.hidden = !shouldShow; // Show or hide.
    }
  }
```

When the right property isn't available or convenient, you can use Element's `attributes` property. This property is a `Map<String, String>`, where the keys are attribute names. For a list of attribute names and their meanings, see the MDN Attributes page. Here's an example of setting an attribute's value:

```
  elem.attributes['someAttribute'] = 'someValue';
```

Creating elements

You can add to existing HTML pages by creating new elements and attaching them to the DOM. Here's an example of creating a paragraph (<p>) element:

```
var elem = ParagraphElement();
elem.text = 'Creating is easy!';
```

You can also create an element by parsing HTML text. Any child elements are also parsed and created.

```
var elem2 = Element.html(
  '<p>Creating <em>is</em> easy!</p>',
);
```

Note that `elem2` is a `ParagraphElement` in the preceding example.

Attach the newly created element to the document by assigning a parent to the element. You can add an element to any existing element's children. In the following example, `body` is an element, and its child elements are accessible (as a List<Element>) from the `children` property.

```
document.body.children.add(elem2);
```

Adding, replacing, and removing nodes

Recall that elements are just a kind of node. You can find all the children of a node using the `nodes` property of Node, which returns a List<Node> (as opposed to `children`, which omits non-Element nodes). Once you have this list, you can use the usual List methods and operators to manipulate the children of the node.

To add a node as the last child of its parent, use the List `add()` method:

```
querySelector('#inputs').nodes.add(elem);
```

To replace a node, use the Node `replaceWith()` method:

```
querySelector('#status').replaceWith(elem);
```

To remove a node, use the Node `remove()` method:

```
// Find a node by ID, and remove it from the DOM.
querySelector('#expendable').remove();
```

Manipulating CSS styles

CSS, or *cascading style sheets*, defines the presentation styles of DOM elements. You can change the appearance of an element by attaching ID and class attributes to it.

Each element has a `classes` field, which is a list. Add and remove CSS classes simply by adding and removing strings from this collection. For example, the following sample adds the `warning` class to an element:

```
var elem = querySelector('#message');
elem.classes.add('warning');
```

It's often very efficient to find an element by ID. You can dynamically set an element ID with the `id` property:

```
var message = DivElement();
message.id = 'message2';
message.text = 'Please subscribe to the Dart mailing list.';
```

You can reduce the redundant text in this example by using method cascades:

```
var message = DivElement()
  ..id = 'message2'
  ..text = 'Please subscribe to the Dart mailing list.';
```

While using IDs and classes to associate an element with a set of styles is best practice, sometimes you want to attach a specific style directly to the element:

```
message.style
  ..fontWeight = 'bold'
  ..fontSize = '3em';
```

Handling events

To respond to external events such as clicks, changes of focus, and selections, add an event listener. You can add an event listener to any element on the page. Event dispatch and propagation is a complicated subject; research the details if you're new to web programming.

Add an event handler using *`element.onEvent.listen(function)`*, where *`Event`* is the event name and *`function`* is the event handler.

For example, here's how you can handle clicks on a button:

```
// Find a button by ID and add an event handler.
querySelector('#submitInfo').onClick.listen((e) {
  // When the button is clicked, it runs this code.
  submitData();
});
```

Events can propagate up and down through the DOM tree. To discover which element originally fired the event, use `e.target`:

```
document.body.onClick.listen((e) {
  final clickedElem = e.target;
  // ...
});
```

To see all the events for which you can register an event listener, look for "onEventType" properties in the API docs for Element and its subclasses. Some common events include:

- change
- blur
- keyDown
- keyUp
- mouseDown
- mouseUp

Using HTTP resources with HttpRequest

Formerly known as XMLHttpRequest, the HttpRequest class gives you access to HTTP resources from within your browser-based app. Traditionally, AJAX-style apps make heavy use of HttpRequest. Use HttpRequest to dynamically load JSON data or any other resource from a web server. You can also dynamically send data to a web server.

Getting data from the server

The HttpRequest static method `getString()` is an easy way to get data from a web server. Use `await` with the `getString()` call to ensure that you have the data before continuing execution.

```
  Future main() async {
    String pageHtml = [!await HttpRequest.getString(url);!]
    // Do something with pageHtml...
  }
```

Use try-catch to specify an error handler:

```
  try {
    var data = await HttpRequest.getString(jsonUri);
    // Process data...
  } catch (e) {
    // Handle exception...
  }
```

If you need access to the HttpRequest, not just the text data it retrieves, you can use the `request()` static method instead of `getString()`. Here's an example of reading XML data:

```
  Future main() async {
    HttpRequest req = await HttpRequest.request(
      url,
      method: 'HEAD',
    );
    if (req.status == 200) {
      // Successful URL access...
```

```
    }
    // ...
  }
```

You can also use the full API to handle more interesting cases. For example, you can set arbitrary headers.

The general flow for using the full API of HttpRequest is as follows:

1. Create the HttpRequest object.
2. Open the URL with either `GET` or `POST`.
3. Attach event handlers.
4. Send the request.

For example:

```
  var request = HttpRequest();
  request
    ..open('POST', url)
    ..onLoadEnd.listen((e) => requestComplete(request))
    ..send(encodedData);
```

Sending data to the server

HttpRequest can send data to the server using the HTTP method POST. For example, you might want to dynamically submit data to a form handler. Sending JSON data to a RESTful web service is another common example.

Submitting data to a form handler requires you to provide name-value pairs as URI-encoded strings. (Information about the URI class is in the URIs section of the Dart Library Tour.) You must also set the `Content-type` header to `application/x-www-form-urlencode` if you wish to send data to a form handler.

```
  String encodeMap(Map data) => data.keys
      .map((k) => '${Uri.encodeComponent(k)}=$
{Uri.encodeComponent(data[k])}')
      .join('&');

  Future main() async {
    var data = {'dart': 'fun', 'angular': 'productive'};

    var request = HttpRequest();
    request
      ..open('POST', url)
      ..setRequestHeader(
        'Content-type',
        'application/x-www-form-urlencoded',
      )
      ..send(encodeMap(data));
```

```
    await request.onLoadEnd.first;

    if (request.status == 200) {
      // Successful URL access...
    }
    // ...
  }
```

Sending and receiving real-time data with WebSockets

A WebSocket allows your web app to exchange data with a server interactively—no polling necessary. A server creates the WebSocket and listens for requests on a URL that starts with **ws://**—for example, ws://127.0.0.1:1337/ws. The data transmitted over a WebSocket can be a string or a blob. Often, the data is a JSON-formatted string.

To use a WebSocket in your web app, first create a WebSocket object, passing the WebSocket URL as an argument:

```
  var ws = WebSocket('ws://echo.websocket.org');
```

Sending data

To send string data on the WebSocket, use the `send()` method:

```
  ws.send('Hello from Dart!');
```

Receiving data

To receive data on the WebSocket, register a listener for message events:

```
  ws.onMessage.listen((MessageEvent e) {
    print('Received message: ${e.data}');
  });
```

The message event handler receives a MessageEvent object. This object's `data` field has the data from the server.

Handling WebSocket events

Your app can handle the following WebSocket events: open, close, error, and (as shown earlier) message. Here's an example of a method that creates a WebSocket object and registers handlers for open, close, error, and message events:

```
  void initWebSocket([int retrySeconds = 1]) {
    var reconnectScheduled = false;

    print("Connecting to websocket");

    void scheduleReconnect() {
      if (!reconnectScheduled) {
        Timer(Duration(seconds: retrySeconds),
```

```
            () => initWebSocket(retrySeconds * 2));
      }
      reconnectScheduled = true;
    }

    ws.onOpen.listen((e) {
      print('Connected');
      ws.send('Hello from Dart!');
    });

    ws.onClose.listen((e) {
      print('Websocket closed, retrying in ' + '$retrySeconds
seconds');
      scheduleReconnect();
    });

    ws.onError.listen((e) {
      print("Error connecting to ws");
      scheduleReconnect();
    });

    ws.onMessage.listen((MessageEvent e) {
      print('Received message: ${e.data}');
    });
  }
```

More information

This section barely scratched the surface of using the dart:html library. For more information, see the documentation for dart:html. Dart has additional libraries for more specialized web APIs, such as web audio, IndexedDB, and WebGL.

For more information about Dart web libraries, see the web library overview.

dart:io - I/O for servers and command-line apps

The dart:io library provides APIs to deal with files, directories, processes, sockets, WebSockets, and HTTP clients and servers.

Important: Only Flutter mobile apps, command-line scripts, and servers can import and use `dart:io`, not web apps.

In general, the dart:io library implements and promotes an asynchronous API. Synchronous methods can easily block an application, making it difficult to scale. Therefore, most operations return results via Future or Stream objects, a pattern common with modern server platforms such as Node.js.

The few synchronous methods in the dart:io library are clearly marked with a Sync suffix on the method name. Synchronous methods aren't covered here.

To use the dart:io library you must import it:

```
import 'dart:io';
```

Files and directories

The I/O library enables command-line apps to read and write files and browse directories. You have two choices for reading the contents of a file: all at once, or streaming. Reading a file all at once requires enough memory to store all the contents of the file. If the file is very large or you want to process it while reading it, you should use a Stream, as described in Streaming file contents.

Reading a file as text

When reading a text file encoded using UTF-8, you can read the entire file contents with `readAsString()`. When the individual lines are important, you can use `readAsLines()`. In both cases, a Future object is returned that provides the contents of the file as one or more strings.

```
Future main() async {
  var config = File('config.txt');
  var contents;

  // Put the whole file in a single string.
  contents = await config.readAsString();
  print('The file is ${contents.length} characters long.');

  // Put each line of the file into its own string.
  contents = await config.readAsLines();
  print('The file is ${contents.length} lines long.');
}
```

Reading a file as binary

The following code reads an entire file as bytes into a list of ints. The call to `readAsBytes()` returns a Future, which provides the result when it's available.

```
Future main() async {
  var config = File('config.txt');

  var contents = await config.readAsBytes();
  print('The file is ${contents.length} bytes long.');
}
```

Handling errors

To capture errors so they don't result in uncaught exceptions, you can register a `catchError` handler on the Future, or (in an `async` function) use try-catch:

```
Future main() async {
  var config = File('config.txt');
  try {
    var contents = await config.readAsString();
    print(contents);
  } catch (e) {
    print(e);
  }
}
```

Streaming file contents

Use a Stream to read a file, a little at a time. You can use either the Stream API or `await for`, part of Dart's asynchrony support.

```
import 'dart:io';
import 'dart:convert';

Future main() async {
  var config = File('config.txt');
  Stream<List<int>> inputStream = config.openRead();

  var lines =
      utf8.decoder.bind(inputStream).transform(LineSplitter());
  try {
    await for (var line in lines) {
      print('Got ${line.length} characters from stream');
    }
    print('file is now closed');
  } catch (e) {
    print(e);
  }
}
```

Writing file contents

You can use an IOSink to write data to a file. Use the File `openWrite()` method to get an IOSink that you can write to. The default mode, `FileMode.write`, completely overwrites existing data in the file.

```
var logFile = File('log.txt');
var sink = logFile.openWrite();
sink.write('FILE ACCESSED ${DateTime.now()}\n');
await sink.flush();
await sink.close();
```

To add to the end of the file, use the optional `mode` parameter to specify `FileMode.append`:

```
var sink = logFile.openWrite(mode: FileMode.append);
```

To write binary data, use `add(List<int> data)`.

Listing files in a directory

Finding all files and subdirectories for a directory is an asynchronous operation. The `list()` method returns a Stream that emits an object when a file or directory is encountered.

```
Future main() async {
  var dir = Directory('tmp');

  try {
    var dirList = dir.list();
    await for (FileSystemEntity f in dirList) {
      if (f is File) {
        print('Found file ${f.path}');
      } else if (f is Directory) {
        print('Found dir ${f.path}');
      }
    }
  } catch (e) {
    print(e.toString());
  }
}
```

Other common functionality

The File and Directory classes contain other functionality, including but not limited to:

- Creating a file or directory: `create()` in File and Directory
- Deleting a file or directory: `delete()` in File and Directory
- Getting the length of a file: `length()` in File
- Getting random access to a file: `open()` in File

Refer to the API docs for File and Directory for a full list of methods.

HTTP clients and servers

The dart:io library provides classes that command-line apps can use for accessing HTTP resources, as well as running HTTP servers.

HTTP server

The HttpServer class provides the low-level functionality for building web servers. You can match request handlers, set headers, stream data, and more.

The following sample web server returns simple text information. This server listens on port 8888 and address 127.0.0.1 (localhost), responding to requests for the path `/dart`. For any other path, the response is status code 404 (page not found).

```
Future main() async {
  final requests = await HttpServer.bind('localhost', 8888);
  await for (var request in requests) {
    processRequest(request);
  }
}

void processRequest(HttpRequest request) {
  print('Got request for ${request.uri.path}');
  final response = request.response;
  if (request.uri.path == '/dart') {
    response
      ..headers.contentType = ContentType(
        'text',
        'plain',
      )
      ..write('Hello from the server');
  } else {
    response.statusCode = HttpStatus.notFound;
  }
  response.close();
}
```

HTTP client

The HttpClient class helps you connect to HTTP resources from your Dart command-line or server-side application. You can set headers, use HTTP methods, and read and write data. The HttpClient class does not work in browser-based apps. When programming in the browser, use the dart:html HttpRequest class. Here's an example of using HttpClient:

```
Future main() async {
  var url = Uri.parse('http://localhost:8888/dart');
  var httpClient = HttpClient();
  var request = await httpClient.getUrl(url);
  var response = await request.close();
  var data = await utf8.decoder.bind(response).toList();
  print('Response ${response.statusCode}: $data');
  httpClient.close();
}
```

More information

This page showed how to use the major features of the dart:io library. Besides the APIs discussed in this section, the dart:io library also provides APIs for processes, sockets, and web sockets. For more information about server-side and command-line app development, see the server-side Dart overview.

For information on other dart:* libraries, see the library tour.

Summary

This page introduced you to the most commonly used functionality in Dart's built-in libraries. It didn't cover all the built-in libraries, however. Others that you might want to look into include dart:collection and dart:typed_data, as well as platform-specific libaries like the Dart web development libraries and the Flutter libraries.

You can get yet more libraries by using the pub package manager. The collection, crypto, http, intl, and test libraries are just a sampling of what you can install using pub.

To learn more about the Dart language, see the language tour.

Chapter III

Layout & Widgets