

PART III



The Dart Language: Basics



The Dart Comment System

In this chapter we will cover the Dart comment system—the different types of comments available when you write your Dart code. We'll see inline, block, and documentation comments and how to use them to properly document your applications. In addition, at the end of the chapter we'll see the markdown syntax you would use to add extra functionality to your programs' documentation.

Comments Overview

As you know, comments are an essential part of the source code of any application. They help to clarify some tricky parts of the code, especially if you have to return to them later to change anything.

They also help to other team members. In development groups where there are many people involved in the same projects, it is essential that each person develops his work, but also facilitates the work of others, leaving clear comments on the construction of the code to avoid any confusion.

Comments are used in all languages, even for something as simple as to auto generate the system documentation. For example, PHP, Java, and JavaScript use PHPDoc, JavaDoc, and JSDoc, respectively. As you know, in Dart we would use docgen for this same purpose.

In any case, we always need to document, so we will learn how to create these types of comments.

Inline Comments

In Dart, we can create comments that extend to a single line by beginning them with the characters `//`.

```
String htmlEscape(String text) {  
    //More efficient implementation.  
    return text.replaceAll("&", "&amp;")  
        .replaceAll("<", "&lt;")  
        .replaceAll(">", "&gt;")  
        .replaceAll("'", "&quot;")  
        .replaceAll("'", "&apos;");  
}
```

In this example we have created an inline comment in a function indicating the reason why we used this type of implementation.

Block Comments

In Dart, as in other programming languages, there are also block comments. We can use them when we need to detail more information about our code. These comments begin with `/*` and are closed with `*/`.

```
/*
 * We modify the method htmlEscape in version 1.2
 * Modified the replacement methods
 * for a more efficient and secure implementation.
 */
String htmlEscape(String text) {
  //More efficient implementation.
  return text.replaceAll("&", "&amp;")
    .replaceAll("<", "&lt;")
    .replaceAll(">", "&gt;")
    .replaceAll("'", "&quot;")
    .replaceAll("'", "&apos;");
}
```

In this example you can see a multiple-line comment is created to add more information. The leading `*` is stylized and is not specifically required, as you can see below. This is a format type, but another format is possible. Below you can see another way to format block comments:

```
/* We modify the method htmlEscape in version 1.2. Modified the replacement
   methods for a more efficient and secure implementation. */
String htmlEscape(String text) {
  //More efficient implementation.
  return text.replaceAll("&", "&amp;")
    .replaceAll("<", "&lt;")
    .replaceAll(">", "&gt;")
    .replaceAll("'", "&quot;")
    .replaceAll("'", "&apos;");
}
```

Documentation Comments

Previously, we learned there are applications for generating system documentation automatically by using the comments in our code. You could improve your application by adding a lot of comments about your system, and in addition you would benefit by getting great system documentation.

You know that in Dart this is possible thanks to the docgen tool, which uses in documentation the comments you insert when you write your code. These documentation comments are similar to block comments, with a small difference—they begin with `/**` and are closed with `*/`, and each line is preceded by `*`.

Unlike above, when we saw the block comments, now the `*` followed by a single space is required at the beginning of each line. This type of comments is shown in Figure 11-1.

```

/**
 * The [Collection] interface is the public interface of all
 * collections.
 */
interface Collection<E> extends Iterable<E> {
  /**
   * Applies the function [f] to each element of this collection.
   */
  void forEach(void f(E element));

  /**
   * Returns a new collection with the elements [: f(e) :]
   * for each element [e] of this collection.
   *
   * Note on typing: the return type of f() could be an arbitrary
   * type and consequently the returned collection's
   * type is Collection.
   */
  Collection map(f(E element));
}

```

Figure 11-1. Documentation block comments

Use the triple slash `///` to create inline documentation comments. The Dart style guide currently recommends the triple slash for documentation comments, even for multi-line comments. This style, shown in Figure 11-2, is the preferred method for documentation.

```

_checkOpen() {
  if (!isOpen) throw new StateError('$runtimeType is not open');
}

/// Returns a Future that completes when the store is opened.
/// You must call this method before using
/// the store.
Future open();

/// Returns all the keys as a stream. No order is guaranteed.
Stream<String> keys() {
  _checkOpen();
  return _keys();
}
Stream<String> _keys();

/// Stores an [obj] accessible by [key].
/// The returned Future completes with the key when the objects
/// is saved in the store.
Future<String> save(V obj, String key) {
  _checkOpen();
  if (key == null) {
    throw new ArgumentError("key must not be null");
  }
  return _save(obj, key);
}

```

Figure 11-2. Documentation inline comments

The documentation comments can appear in any part of your code. Generally, the programmers use them at the beginning of the classes, methods, or functions; however, you could also add documentation comments inside classes, functions, or methods. For example, if you wanted to add documentation comments to any property of a class, you would add them in the appropriate part of your program, as shown in Figure 11-3.

```
/**
 * A parsed URI, such as a URL.
 *
 * **See also:**
 *
 * [URIs][uris] in the [library tour][libtour]
 * [RFC-3986](http://tools.ietf.org/html/rfc3986)
 *
 * [uris]: http://www.dartlang.org/docs/dart-up-and-running/contents/ch0:
 * [libtour]: http://www.dartlang.org/docs/dart-up-and-running/contents/c
 */
class Uri {
  // The host name of the URI.
  // Set to `null` if there is no authority in a URI.
  final String _host;
  // The port. Set to null if there is no port. Normalized to null if
  // the port is the default port for the scheme.
  // Set to the value of the default port if an empty port was supplied.
  int _port;
  // The path. Always non-null.
  String _path;
```

Figure 11-3. Adding documentation comments on instance variables

Comments are also very important at the library level. You can see a lot of inline, block, or even documentation comments placed before the library declaration, as shown in Figure 11-4.

```
//Copyright 2012 Seth Ladd
//
//Licensed under the Apache License, Version 2.0 (the "License");
//you may not use this file except in compliance with the License.
//You may obtain a copy of the License at
//
//    http://www.apache.org/licenses/LICENSE-2.0
//
//Unless required by applicable law or agreed to in writing, software
//distributed under the License is distributed on an "AS IS" BASIS,
//WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//See the License for the specific language governing permissions and
//limitations under the License.

/**
A unified, asynchronous, easy-to-use library for offline-enabled
browser-based web apps. Kinda sorta a port of Lawnchair to Dart,
but with Futures and Streams.

Lawndart uses Futures to provide an asynchronous, yet consistent,
interface to local storage, indexed db, and websql. This library is designed
for simple key-value usage, and is not designed for complex transactional
queries. This library prefers simplicity and uniformity over expressiveness.

You can use this library to help deal with the wide array of client-side
storage options. You should be able to write your code against the Lawndart
interface and have it work across browsers that support at least one of the
following: local storage, indexed db, and websql.

# Example

    var db = new IndexedDbStore('simple-run-through', 'test');
    db.open()
      .then((_) => db.nuke())
      .then((_) => db.save("world", "hello"))
      .then((_) => db.save("is fun", "dart"))
      .then((_) => db.getKey("hello"))
      .then((value) => query('#text').text = value);

See the `example/` directory for more sample code.

*/
library lawndart;
```

Figure 11-4. Comments at the library level

Markdown

Docgen tool also recognizes other Dart Doc comment syntax inspired by the markdown package (<https://pub.dartlang.org/packages/markdown>). This syntax is used by docgen to format documentation comments and add extra capabilities to the documentation, such as links, code snippets, code font, italics, boldface, ordered and unordered list item, and so on.

We'll show you how to use this new functionality to improve the documentation of your own APIs.

Links

You can add links to your Dart Doc comments, which could point to an identifier, constructor, or external hyperlink. `[id]` or `[id](uri)` will generate a hyperlink to a Dart identifier, a class, a method, or a property.

In the coming example we've created two classes, and the Dart Doc for `FirstClass` has links to both `FirstClass` and `SecondClass` identifiers.

```
/// First class is used to create [FirstClass] objects.
/// This class also uses [SecondClass] to accomplish other tasks.
class FirstClass {}

class SecondClass {}
```

After running `docgen` you can view the links in the generated documentation, as shown in Figure 11-5.

Classes

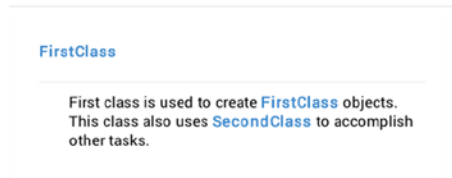


Figure 11-5. Links to Dart identifiers

The links you add into your Dart Doc comments can point to constructor methods or external hyperlinks, as you can see in the code example below. `[new c]` or `[new c](uri)` will create a new link to the constructor for `c`. `[text](uri)` or `<uri>` will create an external hyperlink using text as the link text in the first case, and the `uri` as link text in the second case.

Using these kinds of links in your code will generate the documentation you can see in the code below and in Figure 11-6.

```
/// First class is used to create [FirstClass] objects.
/// This class also uses [SecondClass] to accomplish other tasks.
class FirstClass {

  /// The [new FirstClass] constructor is inspired on
  /// [Uri class](https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.Uri)
  /// You can find this class on <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core>
  FirstClass();

}

class SecondClass {}
```

Constructors

`FirstClass()`

The new `FirstClass` constructor is inspired on [Uri class](#)
 You can find this class on
<https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core>

Figure 11-6. External links

Block Styles

You can use a blank line to denote the end of a paragraph:

```
/// This is a paragraph.
///
/// This is another paragraph.
```

You can add code snippets into your Dart Doc comments using four blank spaces after the line's comment. It means five spaces after the `*` or `/`, as seen below:

```
/**
 *
 * Example:
 *
 *   Uri uri1 = Uri.parse("a://b@c:4/d/e?f#g");
 *   Uri uri2 = uri1.replace(scheme: "A", path: "D/E/E", fragment: "G");
 *   print(uri2); // prints "A://b@c:4/D/E/E/?f#G"
 *
 */
```

You can specify header text using `##` as you can see below and in Figure 11-7:

```
/**
 * ## Other resources
 *
 * See [StringBuffer] to efficiently build a string incrementally. See
 * [RegExp] to work with regular expressions.
 *
 */
```

Other resources

See [StringBuffer](#) to efficiently build a string incrementally. See
[RegExp](#) to work with regular expressions.

..

Figure 11-7. Adding header text to Dart Doc comments

You can also add ordered or unordered lists to your Dart Doc comments using `*` as the first character for unordered lists, or using numbers such as 1., 2., and so on for ordered lists. This is shown in the code below and in Figure 11-8.

```
/// This class is used to create:
///
/// * FirstClass objects
/// * User objects
/// * SecondClass objects
///
/// This class is also used to create:
///
/// 1. Objects with one property
/// 2. Objects with two properties
/// 3. Objects with three properties
///
class FirstClass {}
```

FirstClass class

Extends: `Object`

This class is used to create:

- FirstClass objects
- User objects
- SecondClass objects

This class is also used to create:

1. Objects with one property
2. Objects with two properties
3. Objects with three properties

Figure 11-8. Using ordered and unordered lists

Inline Styles

In our Dart Doc comments we can use inline styles to emphasize notes or warnings as well as use code-style font. Let's see some examples.

``code`` or `[:code:]` will use code-style font for the code you write. Using a single underscore or single asterisk will mark the word or sentence as italics, and double underscore or double asterisks will mark the word or sentence as boldface.

Let's see these inline styles in action in the code below and in Figure 11-9.

```
/// This class is used to create [FirstClass] objects.
///
/// `new FirstClass();` will create a new [FirstClass] object.
///
```

```

/// **Warning** This class uses cache.
///
/// Remember _clear the cache_ after using these objects.
///
class FirstClass {}

```

FirstClass class

Extends: `Object`

This class is used to create `FirstClass` objects.

`new FirstClass();` will create a new `FirstClass` object.

Warning This class uses cache.

Remember *clear the cache* after using these objects.

Figure 11-9. *Inline styles*

Summary

In this chapter we have learned the following:

- What Dart Doc is
- The different types of comments you can use in your applications
- How to create inline comments
- How to create block comments
- How to use the documentation comments
- What Markdown is and how to use it to improve your program's documentation
- How to use links, block styles, and inline styles



Understanding Operators and Expressions

In this chapter we will see what operators and expressions are and the different types we can use in our Dart applications. At the end of the chapter we'll review how to create your own operators.

An Introduction to Dart Operators

Operators are an essential part of any programming language. With them, you can do arithmetic operations with numbers, assign values between variables, and make decisions in a particular moment so that your program will execute one or another block of instructions.

Dart is an object-oriented language; everything in Dart is an object, so the operators are aliases (shortcuts) to methods of instance. These methods are special in that they cannot be assigned to a variable as other objects can, and they have special names, for example, `+`, `-`, `*`, `%`, and so on.

Essentially, an instance method is a classification that is applied to specific methods defined by a class. Instance methods work on object instance variables, but also have access to the class variables.

Do not worry if you have not understood anything so far, as we will see it in more detail when we study classes, methods, instance variables, and object-oriented programming with Dart. The operators, defined by default in Dart, can be overridden or redeclared. If you want to override an operator in one of your classes, you must use the reserved keyword `operator`, as shown in [Figure 12-1](#).

```

abstract class double extends num {
  static const double NAN = 0.0 / 0.0;
  static const double INFINITY = 1.0 / 0.0;
  static const double NEGATIVE_INFINITY = -INFINITY;
  static const double MIN_POSITIVE = 5e-324;
  static const double MAX_FINITE = 1.7976931348623157e+308;

  double remainder(num other);

  /** Addition operator. */
  double operator +(num other);

  /** Subtraction operator. */
  double operator -(num other);

  /** Multiplication operator. */
  double operator *(num other);

  double operator %(num other);

  /** Division operator. */
  double operator /(num other);

```

Figure 12-1. Defining a class with its operators

In Figure 12-1 we can see how to define the double class, which has different operators, or methods with special names: addition operator [+], subtraction operator [-], multiplication [*], division [/], and module operator [%].

■ **Note** Not all operators can be overridden, only a certain subset such as <, +, |, [], >, /, ^, []=, <=, ~/, &, ~, >=, *, <<, ==, -, %, >>

Operator Types

Types of operators in Dart are classified as:

- Arithmetic
- Equality and relational
- Type test
- Assignment
- Conditional
- Bit
- Other

Let us explore these in more detail.

Arithmetic

Below you can see Table 12-1 with the Arithmetic operators.

Table 12-1. *Arithmetic operators*

+	Addition
-	Subtraction
-expr	Unary negation
*	Multiplication
/	Division
~/	Integer division
%	Remainder
++var	Increment before executing other operation
var++	Increment after running the statement
--var	Decrement before executing other operation
var--	Decrement after running the statement

Let's see some examples of arithmetic operators.

```
void main() {
    num a = 3;
    num b = 15;

    print(a + b); // 18
    print(a - b); // -12
    print(-a);    // -3
    print(a * b); // 45
    print(a / b); // 0.2
    print(a ~/ b); // 0
    print(a % b); // 3

    print(a);      // 3
    print(++a);    // 4
    print(a);      // 4
    print(a++);    // 4
    print(a);      // 5

    print(b);      // 15
    print(--b);    // 14
    print(b);      // 14
    print(b--);    // 14
    print(b);      // 13
}
```

Equality and Relational

In Table 12-2 you can see the equality and relational operators.

Table 12-2. *Equality and relational operators*

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Below you can see examples of equality and relational operators in action.

```
void main() {
    num a = 3;
    num b = 15;

    print(a == b); // false
    print(a != b); // true

    print(a > b); // false
    print(a < b); // true

    print(a >= b); // false
    print(a <= b); // true
}
```

Type Test

Type test operators are described in Table 12-3.

Table 12-3. *Type test operators*

as	Type casting, converts an object of one type into an object of another type
is	True if the object is of the indicated type
is!	False if the object is of the indicated type

Here are some examples of type test:

```
void main() {
    int a = 3;

    print((a as num).toDouble()); // 3.0

    print(a is double); // false
    print(a is! double); // true
}
```

Assignment

We'll show you the assignment operators in Table 12-4.

Table 12-4. *Assignment operators*

=	Assigns a value to a variable
*=	Multiplication assignment
~/=	Integer-division assignment
%=	Module assignment
+=	Addition assignment
-=	Subtraction assignment
/=	Division assignment
<<=	Bitwise left-shift assignment
>>=	Bitwise right-shift assignment
&=	Bitwise AND assignment
^=	Bitwise XOR assignment
!=	Bitwise OR assignment

Let's see some examples of assignment operators:

```
void main() {
    var a;

    a = 15;
    print(a); // 15

    a *= 3;
    print(a); // 45

    a ~/= 2;
    print(a); // 22
}
```

```
a %= 3;
print(a); // 1

a += 14;
print(a); // 15

a -= 7;
print(a); // 8

a /= 2;
print(a); // 4.0

a = a.toInt();
a <<= 2;
print(a); // 16

a >>= 1;
print(a); // 8

a &= 0;
print(a); // 0

a ^= 1;
print(a); // 1

a != 1;
print(a); // 1

}
```

Conditional

In Table 12-5 you'll see the conditional operators.

Table 12-5. *Conditional operators*

!expr	Reverses the value of the conditional expression
	Logical OR
&&	Logical AND

Here you can see an example of the use of conditional operators:

```
void main() {
    num a = 3;

    print(a == 3); // true
    print(!(a == 3)); // false

    print(a > 2 || a < 4); // true

    print(a > 2 && a < 4); // true
}
```


Bit

Bit operators are described in Table 12-6.

Table 12-6. *Bit operators*

&	AND
	OR
^	XOR
~expr	Reverses the value of Bit
<<	Left shift
>>	Right shift

Below you can find examples of bit operator usage.

```
void main() {
    print(1 & 0); // 0

    print(0 | 1); // 1

    print(1 ^ 0); // 1

    print(4 << 1); // 8

    print(4 >> 1); // 2
}
```

Others

Dart has other operators commonly used, shown in Table 12-7.

Table 12-7. *Other operators*

()	Call a function
[]	Reference a value from a list
cond?exp1:exp2	If the condition is true, expression 1 is executed; otherwise expression 2 is executed.
.	Accessing the members of an object
..	Making multiple operations on an object, known as cascade operator

An Introduction to Expressions

The expressions in Dart are code snippets. They are evaluated at runtime to get a value, which is always an object. Each expression has a static type associated, and each value has a dynamic type associated.

An expression is a set of data or functions connected by arithmetic, logic, or any other operator.

We will go deep into the data types later, so do not worry too much for now, and let's continue with expressions.

Below we'll describe the different type of expressions you can use in your Dart programs.

Conditional

Evaluates one of two expressions based on a Boolean condition.

```
expr1 ? expr2 : expr3
```

If `expr1` is true, then it executes `expr2`; in any other case, it will execute `expr3`.

Logical

Logical expressions combine Boolean objects with conjunction and disjunction operators.

```
expr1 || expr2
```

```
expr1 or expr2
```

```
expr1 && expr2
```

```
expr1 and expr2
```

Bit

You can manipulate the individual bits of numbers using bit expressions.

```
expr1 & expr2
```

Multiply bits logically, for example, `12 & 10` makes 8.

```
expr1 | expr2
```

Add bits logically, for example, `12 | 10` makes 14.

```
expr1 ^ expr2
```

Exclusive addition of the bits, for example, `12 ^ 10` makes 6.

Equality

This checks equality or identity on objects.

```
expr1 == expr2
```

This expression returns a Boolean object `true` if `expr1` is equal to `expr2` and return `false` in any other case.

```
expr1 != expr2
```

This expression returns a Boolean object `true` if `expr1` is different than `expr2`, and returns `false` otherwise.

Relational

This invokes relational operators on the expression objects.

```
expr1 > expr2
```

Returns `true` if `expr1` is greater than `expr2`, `false` otherwise.

```
expr1 >= expr2
```

Returns `true` if `expr1` is greater than or equal to `expr2`, `false` otherwise.

```
expr1 < expr2
```

Returns `true` if `expr1` is less than `expr2`, `false` otherwise.

```
expr1 <= expr2
```

Returns `true` if `expr1` is less than or equal to `expr2`, `false` otherwise.

Bitwise

Invokes shift operators on the expression objects.

```
expr1 >> expr2
```

Moves a bit pattern to the right, discarding the bits to the left.

```
expr1 << expr2
```

Moves a bit pattern to the left and fills with zeros to the right.

Addition

Invokes addition operators on expression objects.

```
expr1 + expr2
```

```
expr1 - expr2
```

Multiplication

Invokes multiplication operators on expression objects.

```
expr1 * expr2
expr1 / expr2
expr1 ~/ expr2
expr1 % expr2
```

Pre-Expressions and Post-Expressions

Also known as unary expressions, these work on the expression objects. You could get different results depending if you uses pre- or post-expressions.

```
++expr. Increments expr by one and returns expr
expr++. Returns expr and then increments its value by one
--expr. Decrements expr by one and returns expr
expr--. Returns expr and decrements its value by one
```

Assignment

Assigns value to expression objects. With this, you can change the value of a variable or property whenever it is mutable, which means it can be changed.

The main operator is [=], but there are more, for example, [*=], [/=], [~/=], [%=], [+=], [-=], [<<=], [>>=], [&=], [^=], [|=].

`expr1 += 5`. Gets `expr1` value, adds 5, and assigns the new value to `expr1`.

```
void main() {
    var a;

    a = 15;
    print(a); // 15

    a *= 3;
    print(a); // 45

    a ~/= 2;
    print(a); // 22

    a %= 3;
    print(a); // 1

    a += 14;
    print(a); // 15

    a -= 7;
    print(a); // 8
```

```

a /= 2;
print(a); // 4.0

a = a.toInt();
a <<= 2;
print(a); // 16

a >>= 1;
print(a); // 8

a &= 0;
print(a); // 0

a ^= 1;
print(a); // 1

a != 1;
print(a); // 1
}

```

How to Create Your Own Operators

At the beginning, we mentioned that Dart lets you overwrite or redefine operators for any use you need. Let's see an example of how easy it is.

We will create a class `Address` that will store the street where a person lives, and we will use the operator `[+]` to get the full address, including street, number, floor, and city, as shown below.

```

/// Address class
class Address {
  /// Instance variable with the address.
  String _dir;

  /// Simple class constructor.
  Address(this._dir);

  /// Add operator, concatenates our address with more information.
  Address operator + (Address more_info) {
    return new Address("$_dir, $more_info");
  }

  /// Overrides Object.toString() method.
  String toString() => _dir;
}

void main() {
  var adrs = new Address("Main street");
  print(adrs); // Main street

  adrs += new Address("number 3, 4th floor, Madrid");
  print(adrs); // Main street, number 3, 4th floor, Madrid
}

```

In this example we've created our `Address` class in order to create a custom addition operator. This method needs two special things. One of them is that it must return an `Address` object. Second, it must require an `Address` object as a parameter. This is an operator method that operates on `Address` objects, adding two new `Address` objects.

Do not worry about class definition, constructor, overriding, etc. in the example, just look the `Address` operator `+` method.

```
///Add operator, concatenates our address with more information.
Address operator + (Address more_info) {
    return new Address("$_dir, $more_info");
}
```

And, just look at how in the main function we could add two different `Address` objects.

```
adrs += new Address("number 3, 4th floor, Madrid");
```

Summary

In this chapter we have learned the following:

- What the Dart operators are
- What the Dart expressions are
- Some examples combining operators and expressions
- How you can create your own operators



Mastering Dart's Variables and Data Types

In this chapter we will see all of Dart's basic data types such as string, numbers, and Boolean. We will also look at how to define variables in Dart and how to use `final` and `const` reserved keywords.

Later, we will see more complex types such as lists, sets, and maps as well as how to work with regular expressions and `DateTime` objects.

An Introduction to Variables and Data Types

We will start with a very basic and simple definition about what variables are.

The variables are just storage places, memory places that we'll use to store information.

You already know operators are necessary and basic in all programming languages, but without the variables the operators would not make sense. We store strings, numbers, results of other functions, instances, etc. in memory. To access them you must use references, which are the variables. Variables are names that we give to those parts of our computer's memory where data is stored.

If we have access to the data, we can operate with them. If we store number 5 in a variable and number 2 in another variable, we could use the addition operator to work with those variables and add the numbers contained in them, getting a result. This new result could be stored in another variable to use it later if you need it.

Dart also supports different types of data to define the many variables we use in our programs. The data type is like an annotation that you can use so as to know whether the value contained in the variable is a number, a string, or other data type, and thus you can work properly with this data type. For example, you could do mathematics operations such as subtraction or division with numbers but not with strings of text.

Dart uses data types to improve your development experience, warning you about errors while you're writing your applications. Data types also help Dart translate Dart code to JavaScript code.

Let's look at some simple examples using variables:

```
void main() {  
  var variable1 = 5;  
  var variable2 = 2;  
  print(variable1+variable2); // The result is 7  
}
```

We have defined two variables without specifying their data type and have added these variables, displaying the results on screen.

In the next example we work with two different variables, one of which is a `String` variable and the other of which is a variable of type `num`. We'll show a message by interpolating the contents of both variables:

```
void main() {
  String variable1 = 'Welcome to Dart';
  num variable2 = 1.6;
  print('$variable1 $variable2'); // The result is "Welcome to Dart 1.6"
}
```

Variables Definition

As we have seen in the previous examples, there are two ways to define the variables in Dart; the first is with the word `var` followed by the variable name.

```
void main() {
  var my_variable;
  var other_variable = 'This is the content of the variable';
}
```

You can also specify the data type of the variable at definition time, as follows:

```
void main() {
  double my_variable;

  String other_variable = 'This is the content of the variable';
}
```

Therefore, Dart is an optional typed language. If you want, you can specify the type annotations or not and write your applications as you would do in JavaScript.

■ Note Using data type annotations is useful for maintaining your applications. The editor and the compiler use data types to help you in checked mode while you are writing code by autocompleting the code and giving you warnings or errors. If checked mode is disabled then the type annotations will be ignored and will not generate errors or warnings.

Notice that when you define a variable, you can indicate its value, or not. In the first case, the variable has no value assigned, so to Dart the value will be `null`, whatever the data type. In the second case, the variable is initialized with the string of text *"This is the contents of the variable."*

Types of Variables: Final and Const

There are two modifiers by which to change the normal behavior of a variable: `final` and `const`. When writing code, you should determine whether a variable will change during execution. If the variable will never change, you can use `final` or `const` to indicate to Dart that these variables will never change, instead of using `var`. See here:

```
void main() {
  final app_title = 'My first Dar application';
  final String app_subtitle = 'So cool!';
}
```


Notice that you can choose to use the data type annotation when you define the variable with `final`.

What is the difference between `const` and `final`? A variable declared as `final` is initialized only once. However, a variable declared as `const` is considered as a constant in run-time. The global-scope variables, local-scope variables, or class members declared as `final` are initialized the first time they are used. This is called *lazy initialization* and helps applications to start up faster.

For any value that you want to be constant at runtime, use `const` followed by the variable name and the value of the variable. Let's see an example.

We're going to create a small application to calculate the area of a circle. As you may know, the area of a circle can be calculated using this formula, $A = \pi * r^2$. The number *pi* is a mathematical constant, so we'll define it as `const` in our Dart program.

```
void main() {
  const pi = 3.14159265358979;
  var r = 2.59;
  var a = pi * r * r;
  print(a); // 21.07411767954567
}
```

Data Types

Dart has several data types that you can use, and they each have special uses:

- Numbers
- Strings
- Booleans
- Lists (also known as *arrays*)
- Maps (in Python you know them as *dictionaries*)

In Dart, every variable is a reference to an object. As you know, Dart is an object-oriented language, and everything in Dart is an object, even the most basic data type. An object is an instance of a class, which we will discuss later.

When you create this object (instances of classes), you use a constructor method to initialize the object and its class members. Some of the Dart data types have their own builders; for example, to create a list you will use the constructor `new List ()`.

But you can also initialize these special data types with what is called a literal.

```
void main() {
  var string = 'This is a string'; // String Object.

  var number = 15;    // Number Object.

  var correct = true; // Bool Object.

  var list = [1,2,3]; // List Object.

  var map = {          // Map Object. Key-value pairs.
    'name': 'Juan',    // String object for name.
    'age': 29,         // Number object for age.
    'sick': false      // Bool object for sick.
  };
}
```

Numbers

In Dart numbers can be `int` or `double`, which are a subtype of `num`. An integer or decimal numeric value that is assigned to a variable defines that variable as being of type `num`. The `num` type includes most of the operations you will use when working with numbers. Mathematical operations you cannot find in `num` you will find in Dart's library: `math` (discussed in detail later), such as *sin*, *cos*, *pow*, *min*, *max*, and so on.

`int` are numbers without a decimal point.

```
void main() {
  var number = 15;
  var other_number = 159763213;
  var hexadecimal_number = 0x3F;
}
```

`double` are numbers with a decimal point.

```
void main() {
  var double_1 = 15.83;
  var double_2 = 159763213.764398216;
  var double_3 = 1.3e-5;
  var double_4 = -98.72;
}
```

Working with Numbers in Dart

Let's look at some of the methods that we can use when working with numbers in Dart. All of these methods are located in `dart:core`, the main Dart library that offers a great set of predefined functions. This library is automatically imported into all programs.

This library defines three basic classes with which to work with `num`, `int`, and `double` data types. The `num` class is the super class of `int` and `double`, which means that `int` and `double` inherit all the methods that we can find in the `num` class.

Convert a string to a number:

```
int.parse('189'); // returns 189 as int object.
```

`double.parse('15.48');` // returns 15.48 as a double object. These parse methods have their own error handling in case you cannot parse and convert the string to a number. Here's an example:

```
var int_number = int.parse('15');
print(int_number);           // shows 15 as int object.
print(int_number is int);    // shows true;

var error_int_number = int.parse('15€', onError: (_) {return null;});
print(error_int_number);     // shows null because it can not parse the string.
print(error_int_number is int); // shows false.

var double_number = double.parse('15.42');
print(double_number);        // shows 15.42 as double object.
print(double_number is double); // shows true;
print(double_number is num);   // shows true;
```

```
var error_double_number = double.parse('15.42$', (_) {return null;});
print(error_double_number); // shows null because it can not parse the string.
print(error_double_number is double); // shows false.
```

`num.parse` also works, which will first try to convert a string to an integer, and if it fails it'll try to convert to a double, and if that fails then it'll run the `onError` handler, if defined. These methods, with their own error handling, are very useful when trying to collect information from the user and when you need to work with it properly.

If you want to convert a number to string, you can use the `.toString()` method:

```
int num = 5;
String mystring = num.toString();
print(mystring); // shows object String 5.
```

Let's now consider the common methods available on class `num` to work with numbers in Dart. Start with a number like this:

```
num number = -15.84;
```

With these methods you can find out if a number is *finite*, *infinite*, *negative*, or *NaN* (is not a number). The method will always return a bool value of `True` or `False`:

```
print(number.isFinite); // True
print(number.isInfinite); // False
print(number.isNegative); // True
print(number.isNaN); // False
```

If you want to know the sign of a number you can use this method:

```
print(number.sign); // -1.0
```

It returns `-1.0` if the number is less than zero, returns `+1.0` if it's greater than zero, or returns the number itself if it is `-0.0`, `0.0`, or `NaN`.

If you want to get the absolute value of a number, use the following code:

```
print(number.abs()); // 15.84
```

See how to round to the next integer greater than the number:

```
print(number.ceil()); // -15
```

See how to round to the previous integer number lowest than the number:

```
print(number.floor()); // -16
```

Now round to the nearest integer number:

```
print(number.round()); // -16
```

Now return the integer part and discard the decimal part of the number:

```
print(number.truncate()); // -15
```

All of these methods will return an integer number, but the `toDouble` variant will return a double number for these same operations, as follows:

```
print(number.ceilToDouble()); // -15.0
```

```
print(number.floorToDouble()); // -16.0
```

```
print(number.roundToDouble()); // -16.0
```

```
print(number.truncateToDouble()); // -15.0
```

You can also convert a double number to integer, using `toInt`:

```
print(number.toInt()); // -15
```

And you can get a string from a number indicating the decimal part digits:

```
number = 3.14159265358979;
```

```
print(number.toStringAsExponential(6)); // 3.141593e+0
```

```
print(number.toStringAsFixed(4)); // 3.1416
```

```
print(number.toStringAsPrecision(2)); // 3.1
```

Two new methods were introduced on the latest Dart releases: `remainder` and `clamp`. The first one returns the remainder of the truncating division of one number by another. The second one clamps a number so as to be in the range between `lowerlimit`, `upperlimit` that you indicate.

```
int number = 18;
var result = 18.remainder(2.5);
print(result); // shows 0.5
```

```
int number = 18;
print(number.clamp(1, 9)); // 9
print(number.clamp(10, 30)); // 18
print(number.clamp(18, 50)); // 18
```

The `int` class contains the methods we have learned and some others:

```
int number = 15;
```

To find out if a number is even or odd, use the following:

```
print (number.isEven); // False
```

```
print (number.isOdd); // True
```

You can get the bit length of the integer number in binary mode by using the following code:

```
print (number.bitLength); // 4 - binary number 1111
```

You can also convert the number to a string object in the given representation, as follows:

```
print(number.toRadixString(5)); // 30
print(number.toRadixString(2)); // 1111
print(number.toRadixString(16)); // f
```

The double class contains the same methods that we have already learned, and the only difference is the use of a constant to work with this type of numbers:

```
static const double NAN = 0.0 / 0.0;
static const double INFINITY = 1.0 / 0.0;
static const double NEGATIVE_INFINITY = -INFINITY;
static const double MIN_POSITIVE = 5e-324;
static const double MAX_FINITE = 1.7976931348623157e+308;
```

The double data type is contagious, which means that when you perform operations with double numbers you'll get results of type double, as follows:

```
int num1 = 15;
double num2 = 2.0;
print(num1 * num2); // 30.0
```

dart:math Library

Now that we have seen the basic Dart class for working with numbers, let's take a look at the `dart:math` library that allows you to make advanced mathematical calculations with numbers. This library provides functionality to generate random numbers, trigonometric functions, and oft-used math constants such as π and e . To be able to use this library in your applications, you need to import it this way:

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

Besides the functions of `dart:math` we have learned, there are other classes that allow you to work with bidimensional representations of positions and rectangles for 2D graphic designs. Let's see some `dart:math` methods. For mathematical constants to use in your applications, see here:

```
print(E); // 2.718281828459045
print(PI); // 3.141592653589793
```

For trigonometric functions to calculate cosine, sine, tangent, arc cosine, arc sine, and arc tangent use the following:

```
var degrees = 30;
var radians = degrees * (PI / 180); // 0.5235987755982988

print(cos(radians)); // 0.8660254037844387

print(sin(radians)); // 0.49999999999999994
```

```
print(tan(radians)); // 0.5773502691896257
print(acos(radians)); // 1.0197267436954502
print(asin(radians)); // 0.5510695830994463
print(atan(radians)); // 0.48234790710102493
```

To calculate square root, exponential function, logarithmic, and power use the following:

```
print(sqrt(25)); // 5.0
print(exp(3)); // 20.085536923187668
print(log(10)); // 2.302585092994046
print(pow(2, 3)); // 8
```

It also includes functions to calculate the minimum and the maximum of two numbers:

```
print(min(2, 8)); // 2
print(max(15, 10)); // 15
```

Random Numbers

We have learned some of the things we can do using `dart:math`, but there is more; one of them is to work with random numbers using the `Random` class.

Instantiate an object of `Random` type by calling its constructor:

```
var rand = new Random();
```

Then you can generate ten random `int` numbers between 1 and 100 with a code like this:

```
for(var i=0; i<10; i++) {
  print(rand.nextInt(100));
}
```

This code could return a sequence of numbers like this:

```
40, 84, 35, 26, 58, 86, 77, 28, 68, 28
```

With this code you will get a list of ten random `double` numbers between 0 and 1:

```
for(var i=0; i<10; i++) {
  print(rand.nextDouble());
}
```

And this example would return a list of numbers like this:

```
0.35400081214332246, 0.2540705633345569, 0.15263890994501117, 0.265262387358984, 0.9386646648924302,
0.8545231577991513, 0.5948728139929099, 0.04713641533447943, 0.6284813334686553, 0.5992417288687604
```

Random also lets you to generate random bool values:

```
for(var i=0; i<10; i++) {
  print(rand.nextBool());
}
```

And this little snippet of code will return a list of values like this:

true, false, false, false, false, true, false, true, true, false

Strings

Dart strings are sequences of UTF-16 characters. To initialize a string you can use a literal enclosed in double quotes or single quotes, as follows:

```
var str_1 = "Example of string enclosed in double quotes";

var str_2 = 'Example of string enclosed in single quotes';

// You can include a quote type inside another.
var str_3 = 'This is a "string"';

// You can escape quotes with backslash.
var str_4 = 'Escaping \'single quotes\' inside single quotes';
```

Dart supports *string interpolation*, which means that in a string you can include the value of another variable or expression using the format `$variableName` or `${expression}`:

```
var my_number = 4000;

var str_1 = 'Dart is used for more than $my_number people!';

print(str_1); // Dart is used for more than 4000 people!
```

You can also define multi-line strings using triple single quotes or triple double quotes, like you do in Python. Within these strings you can also use interpolation. Note that the new line at the beginning is trimmed off; however, the new line at the end of the triple quote is significant (will display an empty line). See here:

```
var name = 'John';

var template = '''
<html>
  <head>
    <tittle> Dart </tittle>
  </head>
  <body>
    <h1> Welcome to Dart $name !! </h1>
  </body>
</html>
''';
```

Later, we will see the `StringBuffer` class and how efficient it is when used to concatenate strings, but if you need to make a simple string concatenation you can use adjacent strings' literals. No '+' sign is needed—it's valid but not required.

```
var address = 'Picasso Street, 12'
'Postal Code: 28001'
'Madrid'
'Spain';

print(address); // Picasso Street, 12 Postal Code: 28001 Madrid Spain
```

Working with Strings

In `dart:core` you can find the classes you previously learned for working with numbers, but in addition you can find other classes useful for working with other basic data types. In this case, you will see how you can work with strings and all the methods available to the `String` class located in `dart:core`.

Something to keep in mind is that strings are *immutable*—you cannot change a string, but you can create operations on strings and assign the result to another string. See here:

```
var my_string = 'Welcome to Dart';
var new_string = my_string.substring(11, 15);
print(new_string); // Dart
```

Internally, the strings work as lists of characters so you can access a particular position in the string and get the character with the operator `[]`

```
var my_string = 'Welcome to Dart';

print(my_string[0]); // W
```

Strings are sequences of characters (using UTF16) so you can get the code of the character at that position:

```
print(my_string.codeUnitAt(0)); // 87
```

You can also get all the character codes on the string:

```
print(my_string.codeUnits); // [87, 101, 108, 99, 111, 109, 101, 32, 116, 111, 32, 68, 97, 114, 116]
```

You can get the length of the string with this method:

```
print(my_string.length); // 15
```

You can find out if a string ends with a specific character, if it ends with another string, or if it ends with a regular expression. The `endsWith`, `startsWith`, `split`, `replace`, `contains`, `lastIndexOf`, and `indexOf` methods let you specify a simple string or regular expression as an argument:

```
print(my_string.endsWith('t')); // True
print(my_string.endsWith('bye')); // False
```



```
print(my_string.startsWith('Wel'));           // True
print(my_string.contains(new RegExp(r'\bto\b'))); // True
print(my_string.contains(new RegExp(r'[0-9]'))); // False
```

You can discover the position of a string or a particular character in this way:

```
print(my_string.indexOf('Dart')); // 11
print(my_string.indexOf('hello')); // -1
print(my_string.lastIndexOf('a')); // 12
```

You can also find out if a string is empty or not:

```
print(my_string.isEmpty);    // False
print(my_string.isNotEmpty); // True
```

You can use the following method to retrieve only one string fragment. You can specify the optional end index parameter as well:

```
print(my_string.substring(11));    // Dart
print(my_string.substring(11, 13)); // Da
```

And you can remove blank characters at the beginning and end using `trim()`:

```
var my_string2 = 'Dart';
print(my_string2.trim()); // Dart
```

With the arrival of Dart 1.4 two more trim methods were added to the `String` class: `trimLeft()` and `trimRight()`. The first one returns the string without any leading whitespace. The second one returns the string without any trailing whitespace. See here:

```
print(my_string2.trimLeft()); // Dart
print(my_string2.trimRight()); // Dart
```

When you work with strings you probably need to fill them with a certain number of characters, like zeros or whitespaces, to get them in the perfect format. Dart brings to you `padLeft()` and `padRight()` methods to accomplish these tasks. The `padLeft()` method will prepend the character you indicate, or whitespace by default, onto the string one time for each position the length is less than a given width. See the following:

```
var str_num = '12';
print(str_num.padLeft(5, '0')); // 00012
str_num = '37546';
print(str_num.padLeft(5, '0')); // 37546
```

On the other hand, with `padRight()` you can append a character, or whitespaces by default, after the string:

```
var name = 'Moises';
var age = '30';
var row = '| ${name.padRight(20, ' ')}${age.padRight(5, ' ')} |';
print(row); // | Moises_____30__ |
```

With the following methods you can find out whether a string contains a particular character or other string. You can also replace only the first occurrence or all existing in a given string. You can also split the string from a character or string. See the following:

```
print(my_string.contains('hello')); // False

print(my_string.contains('Dart')); // True

print(my_string.replaceFirst('to', 'to the amazing')); // Welcome to the amazing Dart

print(my_string.replaceAll('e', '3')); // W3lcom3 to Dart

print(my_string.split('e')); // [W, lcom, to Dart]
```

Converting the string to lowercase or uppercase works as follows:

```
print(my_string.toLowerCase()); // welcome to dart

print(my_string.toUpperCase()); // WELCOME TO DART
```

Now that we've taken a look at the methods of the `String` class on `dart:core`, we want to show you the great potential of the `StringBuffer` class, which, as we mentioned before, is a very efficient implementation for string concatenation.

To create a new `StringBuffer` you must make a call to the constructor:

```
StringBuffer result = new StringBuffer();
```

Once you have created the object, you can start writing content inside the `StringBuffer` with the `write` method. When you want to show the content of the `StringBuffer` object, you can use `.toString()` method. In the example below it's not necessary because the `print` statement automatically calls the `toString` method:

```
result.write('Welcome');
result.write('to');
result.write('Dart');

print(result); // Welcome to Dart
```

You can even write new lines using the `writeln()` method:

```
result.writeln(); // This method add new line at the end of the string buffer.

result.write('version 1.7');
print(result.toString()); // Welcome to Dart
                        // version 1.7
```

You can get the size of the string buffer and whether is empty or not.

```
print(result.length);    // 27
print(result.isEmpty);   // False
print(result.isNotEmpty); // True
```

You can also write a list of strings (`List<string>`) in the string buffer object using the `writeAll` method:

```
result.writeAll(['. the ', 'new ', 'programming ', 'language']);
print(result.toString()); // Welcome to Dart
                          // version 1.7. the new programming language
```

To clean and empty the `StringBuffer` object you can use the `clear()` method:

```
print(result..clear()); // '' [Empty string]
```

Regular Expressions

We have seen how to work with strings and `StringBuffer`. In previous examples, we learned that some methods can accept strings, characters, or regular expressions. To work with regular expressions in Dart, `dart:core` includes the **RegExp** class. Dart regular expressions have the same syntax and semantics as JavaScript.

■ **Tip** See the specification of JavaScript regular expressions at <http://www.ecma-international.org/ecma-262/5.1/#sec-15.10>

Brackets

Brackets are used to search a range of characters. See Table 13-1.

Table 13-1. *Brackets*

[abc]	Search any character specified between the brackets. You can use a hyphen to specify alphabetical ranges, such as [A-Z].
[^abc]	Search for any character that is not indicated between the brackets.
[0-9]	Search for any digit specified between the brackets.
[^0-9]	Search for any digit that is not indicated between the brackets.
(x y)	Search any of the indicated alternatives.

Metacharacters

Regular expressions use some special characters for searches. See Table 13-2.

Table 13-2. *Metacharacters*

.	Search for a character, except the line termination character or newline character
\w	Search for a letter.
\W	Find a non-word character.
\d	Search for digits.
\D	Find a non-digit character.
\s	Search for whitespace characters.
\S	Search for non-whitespace characters.
\b	Search for matches at the beginning or end of a word.
\B	Search for matches not at the beginning or end of a word.
\0	Search for the NUL character.
\n	Search for the newline character.
\f	Search for the line-feed character.
\r	Search for the carriage-return character.
\t	Search for the tab character.
\v	Search for the vertical tab character.
\xxx	Search for the character specified by the octal number xxx.
\xdd	Search for the character specified by the hexadecimal number dd.
\uxxxx	Search for the Unicode character specified by the hexadecimal number xxxx.

For example, to search for numbers in a string you can use a regular expression like this:

```
var text = 'Welcome to Dart';
var re = new RegExp(r'\d+');
print(re.hasMatch(text)); // False
text = 'Welcome to Dart 1.7';
print(re.hasMatch(text)); // True
```

The constructor method of the `RegExp` class lets you create a new regular expression and determine whether you want it to be case-sensitive or support searching across multiple lines.

```
var re = new RegExp('e');
```

In this sample text you can see how the `RegExp` object is handled:

```
var mytext = 'Welcome Dart';
```

You can find out if the regular expression has matches in the text using the `hasMatch` method:

```
print(re.hasMatch(mytext)); // True
```

The `stringMatch` method will return the first occurrence:

```
print(re.stringMatch(mytext)); // e
```

And `allMatches` lets you work with all the occurrences found. In this particular case, we will iterate over all the matches; we've shown the start and end indexes on the string:

```
re.allMatches(mytext).forEach((m) {  
  print(m.start);  
  print(m.end);  
}); // 1, 2  
    // 6, 7
```

These following methods tell you the regular expression pattern, whether it has multi-line support, and whether it's case sensitive:

```
print(re.pattern);           // 'e'  
print(re.isMultiLine);      // False  
print(re.isCaseSensitive);  // True
```

Booleans

Variables defined as *Boolean* with a `bool` data type usually represent a variable that differs between two statements: *true* or *false*. Boolean variables are defined in Dart with the literal `true` or `false`.

```
var my_bool = true;  
bool other_bool = false;
```

■ **Note** Dart only considers as a true value the `true` value for the `bool` variables; everything else is interpreted as `false` (opposite that which happens in JavaScript).

This should be kept in mind when you work with Dart because you will be forced to check the values you expect. See below for an example of the use of conditional expressions on non-Boolean values.

```
var name = 'Juan';  
if(name) {  
  print(name);  
}
```

If you use a code like that, you will get a Dart error displaying that the variable of type `String` is not a `Bool` subtype for use in a conditional expression. This error occurs because the `name` variable is a string; if this variable were a `bool` variable, this type of conditional expression would be evaluated without errors. However, these types of conditional expressions are very common in JavaScript and work perfectly.

You could change the example and run it like this:

```
var name = 'Juan';
if(name != null && name.isNotEmpty) {
  print(name);
}
```

Working with Bool Values

In `dart:core` you can find the **bool** class that you will use when you want to work with Boolean values. This class has no methods, such as `int`, `num`, or `String` classes. Actually these data types are often used in conditional expressions to determine whether a particular block of code is executed. The only method of the class `bool` is the `toString()` method, which returns a `String` object with the value of the Boolean variable. See the following:

```
bool done = false;
if(done) {
  print(' Well done you've finished!');
} else {
  print(' Sorry, but you have to keep trying !');
}
print(done.toString()); // "false"
```

Lists

One of the most commonly used data structures in all programming languages is the list. In other languages you know them as *arrays*. In Dart, the arrays are collections of objects and are called lists. As you have seen, you can initialize a variable of type `List` with a literal using brackets and commas to separate the values. See here:

```
main() {
  var people = ['Patricia', 'Moises'];

  List pets = ['Benchara', 'Erik', 'Lily'];

  List empty_list = new List();

  // Add elements to empty list.
  empty_list.add('First element');
  empty_list.add(28);

  print(people);    // [Patricia, Moises]
  print(pets);      // [Benchara, Erik, Lily]
  print(empty_list); // [First element, 28]
}
```

In this example, you can see how to create and initialize a list from the literal or by using the constructor `new List()`. We added the `add()` method so you can see how to add elements of any type to a list. Also, you can nest lists, which means that within a list another list or other data type can be included.

As you know, Dart has an *optional typing* system, but if you want even to just define the values that will go into your lists, you can do it using angle brackets and the data type, as in this example:

```
main() {
  List<int> numbers_list = [1, 2, 3, 4, 5];
  List<String> names_list = ['Peter', 'John', 'Mary'];
}
```

Working with the List Class

We will now see all the methods available in `dart:core` for the `List` class, which has four different constructors with which to create Dart lists.

The default constructor allows you to create a list with no elements, and you won't need to indicate the list size:

```
var empty_list = new List();
print(empty_list); // []
```

You can also define a list with a fixed-size element:

```
var empty_list_2_elements = new List(2);
print(empty_list_2_elements); // [null, null]
```

You can create a list of fixed size, and also have it be initialized with a value, a number, a string or a map; for example:

```
var my_filled_list_num = new List.filled(3, 0);
var my_filled_list_str = new List.filled(3, 'A');
var my_filled_list_map = new List.filled(3, {});
print(my_filled_list_num); // [0, 0, 0]
print(my_filled_list_str); // [A, A, A]
print(my_filled_list_map); // [{}, {}, {}]
```

You can use the `List.from` constructor and create a list from another list. This constructor has a parameter so as to create a list of fixed or variable length, allowing the list to grow, or not:

```
var my_list_from_another = new List.from(my_filled_list_str, growable:true);
print(my_list_from_another); // [A, A, A]
my_list_from_another.add('B');
print(my_list_from_another); // [A, A, A, B]
```

Finally, you can use the `List.generate` constructor method to create a list from another by using an iterator and ensuring certain conditions are met. The method passed to the generate constructor receives the index of the element in the list. See a sample list:

```
var other_list = ['/etc/', '/home/pi/'];
```

From this you can create a new list using elements of the first and separating them by a backslash:

```
var my_iterator_list_1 = new List.generate(other_list.length, (e) {
  return other_list[e].split('/');
}, growable: true);
print(my_iterator_list_1); // [[, etc, ], [, home, pi, ]]
```

Or you can create a new list, separating each element by a backslash and discarding empty elements:

```
var my_iterator_list_2 = new List.generate(other_list.length, (e) {
  return other_list[e].split('/').where((e) => e != '').toList();
}, growable: true);
print(my_iterator_list_2); // [[etc], [home, pi]]
```

Besides the constructors, there are other methods for working with lists in Dart. We will work on examples about list management with a simple list, like this:

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

This method shows the length of the list, i.e., how many elements it contains:

```
print(my_list.length); // 3
```

You can also change the length of the list, as follows:

```
my_list.length = 5;
print(my_list.length); // 5
```

And if you want to access a specific position and get its value, you can use brackets that indicate the index:

```
print(my_list [1]); // 2
```

Using brackets, you can also assign values to these positions:

```
my_list [3] = 4;
my_list [4] = 5;

print(my_list); // [1, 2, 3, 4, 5]
```

Or you can use the `add()` method to add items to the list. If the list is non-growable, the `add()` method will throw an error.

```
my_list.add(6);
print(my_list); // [1, 2, 3, 4, 5, 6]
```

There is also a method to add one list to another:

```
my_list.addAll([7, 8, 9]);
print(my_list); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```


You can turn over the list with the reversed method:

```
print(my_list.reversed); // (9, 8, 7, 6, 5, 4, 3, 2, 1)
```

This next method allows you to sort a list:

```
my_list.sort();
print(my_list); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The `.sort()` method can take a function that can be used to manually compare values to determine sorting order. Let's see an example. We have this list of applications with their names and versions:

```
List apps = [
  {'name': 'WordPress', 'version': 4},
  {'name': 'SourceTree', 'version': 2},
  {'name': 'Google Chrome', 'version': 38},
  {'name': 'Safari', 'version': 8},
];

apps.forEach((a) => print(a['name'])); // WordPress
                                       // SourceTree
                                       // Google Chrome
                                       // Safari
```

We can sort them by their names in ascending mode using this compare function:

```
// Sort apps by name ascending
var compare = (a, b) => a['name'].compareTo(b['name']);
apps.sort(compare);

apps.forEach((a) => print(a['name'])); // Google Chrome
                                       // Safari
                                       // SourceTree
                                       // WordPress
```

Now we're going to sort them by their version number in descending mode with another compare function:

```
// Sort apps by version descending
compare = (b, a) => a['version'].compareTo(b['version']);
apps.sort(compare);
apps.forEach((a) => print(a['name'])); // Google Chrome
                                       // Safari
                                       // WordPress
                                       // SourceTree
```

You can also shuffle the list elements randomly with the `shuffle()` method:

```
my_list.shuffle();

print(my_list); // [3, 6, 8, 7, 5, 1, 2, 4, 9]
```

Now let's say you have listed these values:

```
my_list = [3, 6, 8, 7, 5, 1, 2, 4, 9];
```

With the following method you can find the position of item 7. It will return -1 if unable to find that item:

```
print(my_list.indexOf(7)); // 3
```

You can get the last position of an item using `lastIndexOf()`, or -1 if it is not present:

```
print(my_list.lastIndexOf('Z')); // -1
```

The `insert()` method lets you to insert a value in a specific position. In this example we insert at position 0 the value of 9:

```
my_list.insert(0, 9);
print(my_list); // [9, 3, 6, 8, 7, 5, 1, 2, 4, 9];
```

You can also insert a list of values into another list, and place those values in a certain position, with the `insertAll` method:

```
my_list.insertAll(5, [3, 3, 3]);
print(my_list); // [9, 3, 6, 8, 7, 3, 3, 3, 5, 1, 2, 4, 9]
```

The previous two methods also let you insert content into a list by expanding the size of the list.

The following method, `setAll`, allows you to replace values in a list with other values, and to do so at a specific position:

```
my_list.setAll(3, [5, 5, 5]);
print(my_list); // [9, 3, 6, 5, 5, 5, 3, 3, 5, 1, 2, 4, 9]
```

To delete a specific value from a list you can use the `remove` method, indicating the item to delete. Note that we're indicating the element to remove, not the index or position of this element:

```
my_list.remove(6);
print(my_list); // [9, 3, 5, 5, 5, 3, 3, 5, 1, 2, 4, 9]
```

You can also remove an element at a given position using `removeAt`:

```
my_list.removeAt(0);
print(my_list); // [3, 5, 5, 5, 3, 3, 5, 1, 2, 4, 9]
```

You can remove the last element of a list using `removeLast`:

```
my_list.removeLast();
print(my_list); // [3, 5, 5, 5, 3, 3, 5, 1, 2, 4]
```

When you're working with lists, you can remove specific elements of a list using the `removeWhere` method and define the condition that must be met to remove those elements:

```
my_list.removeWhere((e) => e==5);
print(my_list); // [3, 3, 3, 1, 2, 4]
```

On the other hand, by using the method `retainWhere` you can get elements that satisfy the defined condition:

```
my_list.retainWhere((e) => e!=3);
print(my_list); // [1, 2, 4]
```

You can keep a specific part of a list indicating the index range. The end index of a sublist is optional; if it's not specified it will provide a sublist, including everything from the start index to the end of the list:

```
print(my_list.sublist(1, 2)); // [2]
```

The `getRange` method will return an iterable that lets you work with the elements indicated in the index from-to range. Note that in this method, as in the previous one, the final index is excluded from the range returned:

```
print(my_list.getRange(0, 2)); // (1, 2)
```

`setRange` lets you copy iterable values in the from-to range. If you also specify the last parameter (`skipCount`), these values are discarded:

```
print(my_list); // [1, 2, 4]
var my_list2= [7, 8, 9, 6];
my_list.setRange(1, 2, my_list2, 2);
print(my_list); // [1, 9, 4]
```

Delete the elements in the from-to range with the `removeRange` method:

```
my_list.removeRange(1, 2);
print(my_list); // [1, 4]
```

Set the values in the from-to range with the specified value as follows:

```
my_list.fillRange(1, 2, 2);
print(my_list); // [1, 2]
```

Delete the values in the from-to range and insert the values given as follows:

```
my_list.replaceRange(1, 2, [7, 8, 9]);
print(my_list); // [1, 7, 8, 9]
```

In addition, you can convert a list into a dictionary where the keys are consecutive numbers and the values of `Map` are the elements of the list, as follows:

```
var my_map = my_list.asMap();
print(my_map.keys); // (0, 1, 2, 3)
print(my_map.values); // (1, 7, 8, 9)
```

If you want to empty a list you can use `clear`:

```
my_list.clear();
print(my_list); // []
```

Iterable

The methods that we have seen are the built-in methods defined by Dart on `dart:core` for the `List` class, but when you work with lists or sets, there are more methods available because `List` and `Set` implement the `Iterable` interface. This interface is very important because it is also implemented by the classes contained in `dart:collection`.

An `Iterable` is an object that, using an iterator, allows for working with objects one at a time. In other words, it allows you to iterate using the `for-in` loop on objects and then do something with them. Here are some examples of iterables using the `for-in` loop:

```
var my_list = [1, 2, 3];
for(var my_element in my_list) {
  print(my_element); // 1
                      // 2
                      // 3
}

Set my_set = new Set.from([true, false]);
for(var value in my_set) {
  print(value); // true
               // false
}

var my_map = {'1': 'one',
              '2': 'two',
              '3': 'three'};
for(var my_element in my_map.values) {
  print(my_element); // one
                    // two
                    // three
}
```

Let's see all the methods available to work with this type of object. With the `map` method you could obtain a new iterable where each iterable element is replaced by the result of the function executed in `map`:

```
var dicc = {'1': 'one',
            '2': 'two',
            '3': 'three'};
dicc.keys.toList().forEach((k) {
  print('${k} :: String? ${k is String} :: Int? ${k is int}');
});
```

For each key in the dictionary it will show this result:

```
// 1 :: String? true :: Int? false
// 2 :: String? true :: Int? false
// 3 :: String? true :: Int? false
```

Now let's transform this map where its keys are strings to a new map where the keys will be int:

```
var keys = dicc.keys.map((key) => int.parse(key));
dicc = new Map.fromIterables(keys, dicc.values);
dicc.keys.toList().forEach((k) {
  print('${k} :: String? ${k is String} :: Int? ${k is int}');
});
```

The `Map.fromIterables(keys, dicc.values)` method will work as expected because Dart's default map is a `LinkedHashMap`, which guarantees the order of values. For each key of the map you will see this result:

```
// 1 :: String? false :: Int? true
// 2 :: String? false :: Int? true
// 3 :: String? false :: Int? true
```

You can get a new `Iterable` for those items that meet a specific condition using the `where` method:

```
var dicc = {'1': 'one',
           '2': 'two',
           '3': 'three'};
var odd = dicc.keys.where((e) => int.parse(e).isOdd);
print(odd); // (1, 3)
```

The `expand` method lets you to create a new `Iterable` from the elements of the initial `Iterable`, expanding each element in zero or more elements from the result of a function. See here:

```
var queryString = 'arg1=value1&arg2=value2&arg3=value3';
var arguments = queryString.split('&');
print(arguments); // [arg1=value1, arg2=value2, arg3=value3]
arguments = arguments.expand((String arg) => arg.split('=')).toList();
print(arguments); // [arg1, value1, arg2, value2, arg3, value3]
```

If you need to check whether an `Iterable` contains a certain element, you can use the `contains` method:

```
var dicc = {'1': 'one',
           '2': 'two',
           '3': 'three'};

print(dicc.values.contains(4)); // false
print(dicc.values.contains('two')); // true
```

You can also iterate over each element and apply a specific function using `forEach`:

```
dicc.keys.forEach((key) {
  key = int.parse(key);
  print(key); // 1
              // 2
              // 3
});
```

The `reduce` method allows you to reduce a data set to a single value by iterating over the elements and combining them using the given function:

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};
var keys_sum = dicc.keys.reduce((total, element) => total + element);
print(keys_sum); // 6
```

There is a similar method that also allows you to initialize the variable so as to store the result to a specified value.

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};

var keys_sum = dicc.keys.fold(45, (total, element) => total + element);
print(keys_sum); // 51
```

You can get a Boolean value if all items (every) of Iterable meet a certain condition:

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};

var keys = dicc.keys.every((e) => e.isEven);
print(keys); // false

keys = dicc.keys.every((e) => e is int);
print(keys); // true
```

Yet another method allows you to get a Boolean value of true or false if any items (any) of Iterable meet a certain condition:

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};

var keys = dicc.keys.any((key) => key.isOdd);
print(keys); // true

keys = dicc.keys.any((key) => key.isEven);
print(keys); // true
```

Using the `join` method you can convert each element to a string and join all the iterable elements with the specified separator, as follows:

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};

var keys = dicc.keys.join("--");
print(keys); // 1--2--3
```

You can convert an iterable to a list or a set using the `.ToList()` and `.toSet()` methods, respectively:

```
var dict = {1: 'one',
            2: 'two',
            3: 'three'};

print(dict.keys.toList()); // [1, 2, 3]
print(dict.keys.toSet()); // {1, 2, 3}
```

You can find the length of an iterable and whether it's empty or not:

```
var dict = {1: 'one',
            2: 'two',
            3: 'three'};

print(dict.keys.length);    // 3
print(dict.keys.isEmpty);   // false
print(dict.keys.isNotEmpty); // true
```

To get the first and last element you could use `first` and `last` methods, respectively. If you need to get only one element contained in the iterable, you can use the `single` method. If the iterable is empty or has more than a single element, the method will run an exception:

```
print(dict.keys.first); // 1
print(dict.keys.last);  // 3
print(dict.keys.single); // Bad state: More than one element
```

See here how to access the iterable and get a specific element at a given position using the `elementAt` method:

```
print(dict.values.elementAt(1)); // two
```

Using the `take` method you can retrieve a specified number of elements:

```
var dict = {1: 'one',
            2: 'two',
            3: 'three'};

print(dict.values.take(2)); // (one, two)
```

You could retrieve a certain number of items using `takeWhile`, which requires the indication of one condition. This method will stop when the condition is not met anymore. See the following:

```
print(dict.keys.takeWhile((k) => k.isEven)); // ()
print(dict.keys.takeWhile((k) => k.isOdd));  // (1)
```

You can keep some elements of the object, but you can also discard some elements:

```
var dict = {1: 'one',
            2: 'two',
            3: 'three',
            4: 'four'};
```

```
print(dicc.values.skip(2));           // (three, four)
print(dicc.keys.skipWhile((k) => k.isEven)); // (1, 2, 3, 4)
print(dicc.keys.skipWhile((k) => k.isOdd));  // (2, 3, 4)
```

Finally, you can keep the first, the last, and the only element that meets a specific condition with these methods:

```
var dicc = {1: 'one',
            2: 'two',
            3: 'three'};

print(dicc.keys.firstWhere((k) => k.isOdd)); // 1
print(dicc.keys.lastWhere((k) => k.isOdd));  // 3
print(dicc.keys.singleWhere((k) => k.isEven)); // 2
```

Sets

Let's look at the Set data type, which lets you work with data sets and operate them as if they were lists. The main features of sets are that they:

- do not allow storage of duplicate elements, and
- are messy

Sets inherit from iterables and share all the methods just learned in the previous section. Let's look at some examples of sets and how to work with them.

To create a new set, we can use any of their constructors; for example, `new Set()` or `Set.from()`.

```
var con = new Set();
con.addAll([1, 1, 2, 3]);
print(con); // {1, 2, 3}

con = new Set.from([1, 1, 2, 2, 3, 3, 4]);
print(con); // {1, 2, 3, 4}
```

■ **Note** When creating the set, duplicate items will disappear.

When working with sets you can check if they contain a specific item using the `contains` method, or if it contains a set of elements by using `containsAll`:

```
print(con.contains(1)); // true
print(con.contains(5)); // false

print(con.containsAll([2, 3])); // true
```

You can look for a specific value and, if the value is within the set, the Set class will return the found element or null if it's not found. See here:

```
print(con.lookup(9)); // null
print(con.lookup(4)); // 4
```


To add items to the set you can use `add` or `addAll`. They work the same way as we've previously seen working with lists. See here:

```
con.add(5);
print(con); // {1, 2, 3, 4, 5}

con.addAll([6, 7, 8, 9]);
print(con); // {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

To remove items from a set you can use the following methods. You can delete a single item, delete multiple items, or delete items that meet a specific condition.

```
con.remove(9);
print(con); // {1, 2, 3, 4, 5, 6, 7, 8}

con.removeAll([8, 7, 6]);
print(con); // {1, 2, 3, 4, 5}

con.removeWhere((e) => e==5);
print(con); // {1, 2, 3, 4}
```

You can return only some elements of a set using `retainAll` followed by an `Iterable`, or you can use `retainWhere` followed by a condition to get a new set that has only those elements specified:

```
con.retainAll([1,2,3]);
print(con); // {1, 2, 3}

con.retainWhere((e) => e.isOdd);
print(con); // {1, 3}
```

Finally, this class offers you the methods of intersection, union, and difference necessary to work with data sets:

```
var con = new Set.from([1, 3]);
var con2 = new Set.from([3, 4]);

print(con.intersection(con2)); // {3}
print(con.union(con2));        // {1, 3, 4}
print(con.difference(con2));    // {1}
```

Maps

A `Map` is an object that associates keys and values; if you know Python, this will be familiar because it is similar to a Python dictionary and very similar to JavaScript Objects too.

In this case a map can be defined by a string literal or by using the constructor `new Map()`. If you use a string to initialize, the keys and values can be of any data type (including lists or other maps). Let's see an example:

```
var literal_map = {
  1: 'values',
  'key': 12,
  true: 'true',
```

```
'false': false,
12.45: 12,
[1, 2]: [1, 2],
{1: 'one'}: {2: 'two'}
};
```

Let's see how to add more key-value pairs to a map:

```
var p1 = {
  'name': 'John',
  'age': 29
};
// You can add Key-Value pair to a map this way.
p1['surname'] = 'Smith';

Map p2 = new Map();
p2['name'] = 'Peter';
p2['age'] = 32;

Map results = new Map();
results [1] = 'Ranked first 1500 points';
results [2] = 'Ranked second 990 points';
results [3] = 'Ranked third 786 points';

// Access to the keys of map.
print(p1['name']);    // John
print(p1['surname']); // Smith

print(p2['old']);     // 32
print(p2[surname]);  // null
```

You can add more keys to a map at any time in your application, and if you try to access an undefined property of a map, Dart will return null.

Just as with lists, you can specify the data type of the key values and the map values using angular brackets:

```
Map<String, double> champions = {
  'Sebastian Vettel': 397.0,
  'Fernando Alonso': 242.0,
  'Mark Webber': 199.0
};

Map<int, List> positions = {
  1: [1, 3, 4, 1, 7, 8],
  2: [4, 13, 1, 2, 4, 5],
  3: [7, 8, 9, 10, 2, 4]
}
```

The Map class has several methods to work with, some of which you should know already. Let's look at them now.

In addition to the constructor methods we've just discussed, the `Map` class has constructors by which to create new maps from other maps or from iterables. See here:

```
var p1 = {
  'nombre': 'John',
  'age': 29
};

var p2 = new Map.from(p1);
p2['age'] = 32;

print(p1);
print(p2);

var my_list = ['one', 'two', 'three', 'four'];

var my_map = new Map.fromIterable(my_list,
  key: (item) => item.toString().length,
  value: (item) => item);

print(my_map); // {3: two, 5: three, 4: four}

var keys = [1, 2, 3];
var values = ['one', 'two', 'three'];
var my_map2 = new Map.fromIterables(keys, values);
print(my_map2); // {1: one, 2: two, 3: three}
```

You can see if it contains a specific key or value with these methods.

```
var my_map = {1: 'one', 2: 'two', 3: 'three'};
print(my_map.containsKey('7')); // false
print(my_map.containsKey(2)); // true

print(my_map.containsValue('7')); // false
print(my_map.containsValue('three')); // true
```

The `Map` class has a method to add a key if it does not exist on the object; this method simultaneously associates a value:

```
my_map.putIfAbsent(4, () => 'four');
print(my_map); // {1: one, 2: two, 3: three, 4: four}
```

It also lets you add other key-value pairs with the `addAll` method:

```
my_map.addAll({5: 'five', 6: 'six'});
print(my_map); // {1: one, 2: two, 3: three, 4: four, 5: five, 6: six}
```

You can delete a key from the map object using `remove`. The key and the associated value are removed. Note that you must indicate the *key* to remove, not the index, position, or element.

```
my_map.remove(5);

print(my_map); // {1: one, 2: two, 3: three, 4: four, 6: six}
```

You can also apply a function to each key-value pair with `forEach`:

```
my_map.forEach((k, v) {
  print('Key ${k} = ${v}'); // Key 1 = one
                                // Key 2 = two
                                // Key 3 = three
                                // Key 4 = four
                                // Key 6 = six
});
```

On the `Map` class there are several methods by which to know the keys and values of the object, the length of key-value pairs, and whether the map is empty or not:

```
print(my_map.keys);      // (1, 2, 3, 4, 6)
print(my_map.values);    // (one, two, three, four, six)
print(my_map.length);    // 5
print(my_map.isEmpty);   // false
print(my_map.isNotEmpty); // true
```

Finally, you can completely empty a map with the `clear` method:

```
my_map.clear();
print(my_map); // {}
```

Date and Time

In addition to the basic Dart data types, we have learned some of the most interesting `dart:core` classes, such as `Lists`, `Sets`, `Maps`, `RegExp`, and so forth. These classes allow you to work with other very useful data structures.

Now let's see a class that you will use very often when you need to work with dates and times: the `DateTime` class. This class is located on `dart:core` and has a set of constructors to define `DateTime` objects.

You can define a `DateTime` object with the current time with the constructor `DateTime.now()`:

```
var my_date = new DateTime.now();
print(my_date); // 2014-10-07 19:31:17.610
```

You can define it explicitly by indicating the year, month, day, hour, minute, second, and millisecond. If a value is not specified, the default is initialized to 0:

```
my_date = new DateTime(2014, 3, 14, 17, 30, 0);
print(my_date); // 2014-03-14 17:30:00.000
```

This class also lets you define `DateTime` objects from strings that fit the ISO 8601 format:

```
my_date = DateTime.parse('2014-12-09 09:00:00');
print(my_date); // 2014-12-09 09:00:00.000
```

You can define `DateTime` objects directly in universal time with `DateTime.utc()` constructor:

```
my_date = new DateTime.utc(2014, 10, 7, 17, 30, 0);

print(my_date); // 2014-10-07 17:30:00.000Z
print(my_date.isUtc); // true
```

Or you can define them from a timestamp indicating milliseconds:

```
my_date = new DateTime.fromMillisecondsSinceEpoch(1394280000000);
print(my_date); // 2014-03-08 13:00:00.000
```

Note that the time in the comment above may vary from computer to computer due to time zones, unless the `isUtc` optional parameter is passed to indicate the time zone, as below:

```
my_date = new DateTime.fromMillisecondsSinceEpoch(1394280000000, isUtc: true);
print(my_date); // 2014-03-08 12:00:00.000Z
```

Besides these `DateTime` constructor methods, this new class offers you getters and methods so as to work with dates. To obtain the values of the `DateTime` object you can use these getters. As you can see, the days of the week in Dart begin on 1 for Monday and end at 7 for Sunday:

```
var today = new DateTime.now(); // 2014-03-08 18:52:33.906
print(today.year);              // 2014
print(today.month);             // 3
print(today.day);              // 8
print(today.hour);             // 18
print(today.minute);           // 52
print(today.second);           // 33
print(today.millisecond);      // 906
print(today.weekday);         // 6 - Saturday
```

You can get local time or universal time from a `DateTime` object in this way:

```
print(today.toLocal()); // 2014-03-08 18:52:33.906
print(today.toUtc());  // 2014-03-08 17:52:33.906Z
```

And you can convert the `DateTime` object into an `String` object with the `toString()` method:

```
print(today.toString()); // 2014-03-08 18:52:33.906
```

We will now define another `DateTime` object and see the operations we can do with it:

```
var yesterday = new DateTime(2014, 3, 7, 18, 23, 0);
```

We can compare both objects easily in this way. You'll get a Boolean value indicating if a date-time is equal to, less than, or greater than another. See here:

```
print(today==yesterday);           // false
print(today.isAtSameMomentAs(today)); // true
print(today.isBefore(yesterday));  // false
print(today.isAfter(yesterday));   // true
```

`DateTime` implements other methods that enable comparisons between date-time objects and then returns `int` values as a result, as follows:

```
print(today.compareTo(yesterday)); // 1
print(yesterday.compareTo(today)); // -1
print(today.compareTo(today));     // 0
```

When working with `DateTime` objects, it is common to add or subtract dates-hours or calculate differences between `DateTime` objects to accomplish some tasks on your programs. All these operations can be easily done in Dart, but before we see them we have to talk about another new class also available in `dart:core` called `Duration`. This class is used as a parameter, or it can be returned as a result indicating differences between dates.

The `Duration` class has its own methods and operators to compare, convert to `String` objects, and make operations such as add, subtract, multiply or divide. This class is used by other parts of the Dart library, such as `Timers` and `Futures`.

The most interesting aspect of the `Duration` class is its constructor, which defines a `Duration` object from the specified parameter; if none is specified, the default will be 0. Let's see an example:

```
var dur = new Duration(days: 20, hours: 4, minutes: 2, seconds: 17, milliseconds: 300,
microseconds: 155897);

print(dur); // 484:02:17.455897
```

You can get the value in days, hours, minutes, seconds, milliseconds, or microseconds with these methods:

```
print(dur.inDays);           // 20
print(dur.inHours);          // 484
print(dur.inMinutes);        // 29042
print(dur.inSeconds);        // 1742537
print(dur.inMilliseconds);   // 1742537455
print(dur.inMicroseconds);   // 1742537455897
```

If you want to add a day to a date, you must use the `.add()` method and a `Duration` object as the parameter indicating the number of days to add. According to our previous `today` variable:

```
print(today.add(new Duration(days:1))); // 2014-03-09 18:52:33.906
```

You can also add hours by indicating as the parameter a `Duration` object expressed in hours:

```
print(today.add(new Duration(hours:10))); // 2014-03-09 04:52:33.906
```

You already know how to add days, hours, or whichever parameter you specify with your `Duration` objects. Now let's see how you can subtract. The process is very similar; you can subtract a specific value of a date with the `subtract()` method, using a `Duration` object as the parameter containing the desired value to subtract—for example, days or hours. See here:

```
print(today.subtract(new Duration(days:1))); // 2014-03-07 18:52:33.906
print(today.subtract(new Duration(hours:10))); // 2014-03-08 08:52:33.906
```

Note that the add and subtract methods return a new instance of the `DateTime` object and do not modify the existing instance.

If you want to know the difference between two dates, you can use the `difference()` method; you'll get a `Duration` object as the result. You can use the `Duration` object as needed, for example, in hours, seconds, or days, with the corresponding getter method. See the following:

```
var today = new DateTime(2014, 10, 9, 18, 23, 0);
var yesterday = new DateTime(2014, 10, 7, 18, 23, 0);
print(today.difference(yesterday).inDays); // 2
print(today.difference(yesterday).inHours); // 48
print(today.difference(yesterday).inSeconds); // 172800
```

Now let's look at two other methods for the `DateTime` class: one of them to get the name of the client time zone and the other to get the offset, which will return a `Duration` object. See here:

```
print(today.timeZoneName); // CET
print(today.timeZoneOffset.toString()); // 1:00:00.000000
```

Summary

In this chapter we have learned:

- About all the basic Dart data types: string, numbers, and Boolean.
- About all the methods available on Dart SDK to work with string, numbers, and Boolean.
- How to define variables in Dart and how to use `final` for lazy initialization and `const` to define constants.
- About more complex Dart types, such as lists, sets, and maps.
- About all the methods you can use to work with lists, sets, and maps.
- About the regular expression objects and how they are similar to JavaScript specifications.
- About `DateTime` objects and how to add, subtract, or calculate difference between dates.



Flow Control Statements

In this chapter we'll see the flow control statements used in our programs to change the execution flow or to repeat sentences. We'll see `if-else` and `switch` statements.

Later we'll see the loops statements in Dart `for` loop and `while` loop that allow you to repeat blocks of code in your applications. We'll also see `for-in` and `forEach` loops that work with collections.

As you know, a program is a sequence of statements executed sequentially. There are two types of statements:

- **Simple statements:** run one after another
- **Control statements:** allow you to change the flow of program execution by introducing cycles and conditions that must be met in order for some blocks of code to be executed.

We'll start by talking about conditional statements, a type of control statement.

If and Else

The `if` statement lets you choose whether a block of instructions is executed or not. If this statement includes an `else` block, then this statement will execute one block of instructions if the condition is `true` and a different one if the condition is `false`. Dart allows `if` statements with optional `else` statements.

The peculiarity is that with Dart, we must be more explicit when we work with conditions and `if` statements, because Dart believes that all values different from `True` are `False`. This is different in JavaScript and, as we discussed in the “Mastering Dart’s Variables and Data Types” chapter, if you use these types of conditions you will get an error.

```
var name = 'John';
if(name) {
  print(name);
}
```

Let's see how to use `if-else` conditional statements in Dart, and per the standard style recommendation, we should always use explicit curly brackets.

```
var weather = 'risk';
if(weather == 'risk') {
  caution();
} else {
  withoutCaution();
}
```


Dart also supports nested if-else-if statements, so you can use them in this way:

```
var color = '#0000ff';
if(color == '#ff0000') {
  print('Red');
} else if(color == '#00ff00') {
  print('Green');
} else if(color == '#0000ff') {
  print('Blue');
} else {
  print('I dont know this color!');
}
```

This type of structure is not the perfect solution, because to do this we must use the switch statement, which is faster than an if-else-if chain and generates a more readable code, as we'll see below.

Switch Statement

As mentioned, the switch statement allows one to make decisions between multiple alternatives, thus avoiding nested if-else-if statements. This improves the readability of the code, and it is a structure that executes faster than using nested if-else-if. Let's see an example of a switch statement.

```
var action = 'Open';
```

```
switch (action) {
  case 'Paid':
    pay();
    break;

  case 'Unpaid':
    unpay();
    break;

  case 'Rejected':
    rejected();
    break;

  default:
    check();
}
```

As you can see, this statement defines some matching cases (case) and some actions to take in case one of them is fulfilled. Having met the case and executed the action, you should always include the break statement to exit switch.

■ **Note** In other languages, not including the break statement after each case will result in strange or inappropriate behavior. In Dart, while you're writing your code you can see how Dart Editor warns you about missing break statements. Even if you try to run your code, it'll throw an error and will stop the execution.

Optionally you can specify the **default** case, which will be executed if none of the other options coincide. In this particular case it is not necessary to include the `break` statement because there are no more cases after it.

The `switch` statement can contain empty cases, as you can see below in the case of *Unpaid*.

```
switch (invoiceStatus) {
  case 'Paid':
    pay();
    break;

  case 'Unpaid':
  case 'Rejected':
    unpaid();
    break;

  default:
    throw(' Error invoice Status !');
}
```

That means that the function `unpaid()` will be executed if the invoice status is `Unpaid` or `Rejected`.

A case of coincidence can contain calls to other functions, as you saw in the previous examples, but also can contain any block of code you want to execute. You can even define local variables that you need. These local variables will only be visible within the scope of that case.

```
switch (invoiceStatus) {
  case 'Paid':
    pay();
    break;

  case 'Paid':
  case 'Rejected':
    unpaid();
    break;

  default:
    var error = 'Error Invoice Status !';
    sendEmail(subject:error, to:'error@company.com');
    throw(error);
}
```

For Loop

Now that we've reviewed conditional statements, it's time to see looping statements. A loop is used to repeat a given sentence or sentences a specified number of times. Dart has several different types of loops, but for now we focus on the `for` loop. This loop has a very familiar format to you because it inherits the syntax of JavaScript, C, and so on.

```
for (var i=0; i < 3; i++) {
  print(i); // 0
            // 1
            // 2
}
```

The for loops also allow you to iterate over List, Map, and Set objects.

```
var my_list = ['one', 'two', 'three', 'four'];

for (var i=0; i < my_list.length; i++) {
  print('$i :: ${my_list[i]}'); // 0 :: one
                                // 1 :: two
                                // 2 :: three
                                // 3 :: four
}
```

Actually, if the object that you go through is a collection, you can use the `forEach()` method. This is a perfect choice if you don't need to know the current iteration count.

```
var my_list = ['one', 'two', 'three', 'four'];
my_list.forEach((item) {
  print(item); // one
               // two
               // three
               // four
});
```

To iterate over a collection, or any iterable, without knowing the iteration count, you can use a `for-in` loop.

```
var documents = ['orders', 'invoices'];
for (var doc in documents) {
  print(doc); // orders
              // invoices
}
```

You could use this `for-in` loop over any Iterable.

While Loop

The while loop repeats a block of code while a condition remains true. This statement will evaluate the condition before entering the loop, and will repeat it while the condition is true.

```
while (missingCalls()) {
  blinkLight();
}
```

If you want to evaluate the condition after executing the loop cycle (i.e., execute the loop at least once), you can use `do-while` statement.

```
do {
  blinkLight();
} while (missingCalls());
```

When you are working with loops you can also use these two statements: `break` and `continue`. These statements will serve to control the execution of the loop.

Break is used to stop the execution of the loop:

```
var i = 100;
while(true) {
  i -= 5;
  print(i);
  if(i < 75) {
    break;
  }
}
```

This example will return this result:

```
95
90
85
80
75
70
```

However, continue allows you to jump to the next iteration of the loop:

```
for (var i = 100; i > 0; i--) {
  if (i.isEven) {
    continue;
  }

  print(i);
  if (i < 75) {
    break;
  }
}
```

This version will show a result like this:

```
99
97
95
93
91
89
87
85
83
81
79
77
75
73
```

Summary

In this chapter we have learned the following:

- About flow control statements
- About if-else and switch statements
- How to use loops to repeat sentences using For or While loops
- How to use for-in and forEach loops when we're working with collections



Working with Functions

In this chapter we'll talk about functions in Dart: how you can create them, how to define them, and which types of parameters they can receive.

We'll see in detail the difference between positional arguments and named arguments. We'll also see default values for the function parameters. Later we'll see the returned values for functions, and at the end of the chapter we'll look at recursive functions in Dart. So far, you've seen single statements, control statements, and loop structures. With these tools you can start to build an application in Dart, but as the application grows up you will need to organize your code better and reuse code you've written elsewhere in the application. Thanks to the functions, you could do this type of thing and get a cleaner, more efficient, and easier maintained code.

A function is a set of instructions that performs a specific task. Functions are executed by calling them from another part of the code, a class method, or other function. Functions can be called as many times as you need and can even call themselves (recursive functions).

Functions can work with parameters and can return results. The function parameters in Dart may be required or optional. If function parameters are optional, they may be optional by their position, known as positional arguments, or optional by name, known as named arguments.

We usually use functions to split a large task into simpler operations or to implement operations that we will use repeatedly in our application.

Defining Functions

Let's see how to define functions in Dart.

```
function_name(function_arguments) {  
  // Body of the function  
}
```

Here you can see a very simple function example:

```
greet(name) {  
  print('Welcome to Dart $name');  
}
```

If you want to execute this function from your main program, you only need to call it, like this:

```
void main() {  
  greet('Peter'); // Executing our sample function.  
}
```

Although the above code is perfectly valid, it is recommended to use the Dart code style guide. Therefore, you should always specify the return data type of the function and the data types of the parameters passed to the function when you define your functions. Thus, our function declaration should look like this:

```
return_type function_name(argument_type argument) {
    // body of the function.
}
```

Continuing with our example, let's change the greeting:

```
void greet(String name) {
    print('Welcome to Dart $name');
}
```

To properly declare a function in Dart and following the code style guide you must specify:

- Function return type or void if the function did not return any value.
- Function name.
- Function parameters specifying data type and name of each parameter.
- Body of the function, between curly braces.

To show you the return data type and how a function will return any value, our sample function will be redeclared as follows:

```
String greet(String name) {
    var msg = 'Welcome to Dart $name';
    return msg;
}
```

In cases of simple functions that have only a single statement, you could use the abbreviated definition. Here is an example:

```
void welcome() => print('Welcome to Dart!');
```

And then you could use this function by calling it like this:

```
void main() {
    welcome(); // Welcome to Dart!
}
```

As we previously mentioned, any function could receive required or optional parameters. Required parameters are indicated between parentheses. You could specify as many as you need, and every argument would be separated by a comma.

```
void greet(String required_parameter_1, num required_parameter_2) {
    // Function body
}
```

Optional parameters are also indicated between parentheses, but in this case we have to indicate them according to a specific nomenclature. As we know, optional parameters can be of two different types: optional by position (positional arguments) or optional by name (named arguments).

Positional Optional Parameters

Positional optional parameters are enclosed between square brackets and follow the same description as the required parameters: data type and parameter name.

```
void greet(String name, [String lastname, num age]) {
  if(lastname != null && age != null) {
    print('Welcome to Dart $name $lastname you are $age years old.');
```

```
  } else {
    print('Welcome to Dart $name.');
```

```
  }
}
```

If you make some calls to this function with these different parameters you could get these results:

```
void main() {
  greet('Peter');           // Welcome to Dart Peter.
  greet('Peter', 'Smith');  // Welcome to Dart Peter.
  greet('Peter', 'Smith', 28); // Welcome to Dart Peter Smith you are 28 years old.
}
```

What would happen if we changed the position of the arguments when we call the function? We would have unexpected results:

```
void main() {
  greet('Smith', 'Peter');  // Welcome to Dart Smith.
  greet('28', 'Peter', 28); // Welcome to Dart 28 Peter you are 28 years old.
  greet(28, 'Peter', 'Smith'); // This will throw an exception.
}
```

What would happen if our function had 150 parameters? It would be almost impossible to not make mistakes when you call your function, right? To handle those situations, and as a means of further API self-documentation, Dart introduced optional parameters which are called by their name.

Named Optional Parameters

In addition to positional optional parameters, we have named optional parameters. They really are very useful because you do not need to call the function in the same way as with positional parameters, and this kind of parameter improves the documentation of the function when the call is made, mainly if the function has a lot of parameters.

Another important benefit is that for positional parameters, if you want to pass the last parameter, all preceding parameters must also be passed. Named parameters allow only the used parameters to be passed.

In our next sample code you can see our function does not have any required arguments, so all the arguments would be optional; in this particular case, they would be called optional arguments.

```
void greeting({String name, String lastname, num age}) {
  var greet = new StringBuffer('Welcome to Dart');
  if (name != null) {
    greet.write(' $name');
```

```
  }
}
```



```

    if (lastname != null) {
        greet.write(' $lastname');
    }

    if (age != null) {
        greet.write(' you are $age years old. ');
    }
    print(greet.toString());
}

void main() {
    greeting(name: 'Peter'); // Welcome to Dart Peter
    greeting(lastname: 'Smith', name: 'Peter'); // Welcome to Dart Peter Smith
    greeting(lastname: 'Smith', age: 29, name: 'Peter'); // Welcome to Dart Peter Smith you are 29 years old.
    greeting(); // Welcome to Dart
    // If we want to pass only the last parameter we can do easily without
    // passing preceding parameters.
    greeting(age: 29); // Welcome to Dart you are 29 years old.
}

```

Note that the order of the parameters passed to the function is not important, because you are mapping the parameter name with the value. This is something really great, because in functions that can accept many optional parameters, remembering the exact order in which to pass the parameters to the function is a task that is heavy and ripe for errors.

If we call our functions through named optional arguments, we can forget the exact parameter order. Furthermore, we would have even better documentation if we used very descriptive names for the optional parameters. This would improve our code-maintenance process.

Default Values

The functions with optional parameters also can get default values. A default value is assigned to the optional parameter if the parameter is not specified when you execute the function. The default value must be specified when you declare your function.

To specify the default value of an optional parameter, we use the equal sign (=) or a colon (:) depending on if it is an optional parameter by position or optional by name.

This is an example of a default value for a positional optional parameter:

```

void greeting([String name= 'ANONYMOUS']) {
    var greet = new StringBuffer('Welcome to Dart');
    if (name != null) {
        greet.write(' $name');
    }
    print(greet.toString());
}

void main() {
    greeting(); // Welcome to Dart ANONYMOUS
    greeting('John'); // Welcome to Dart John
}

```

And here we see an example of a default value for named optional arguments:

```
void greeting({String name: 'ANONYMOUS'}) {
  var greet = new StringBuffer('Welcome to Dart');
  if (name != null) {
    greet.write(' $name');
  }
  print(greet.toString());
}

void main() {
  greeting();           // Welcome to Dart ANONYMOUS
  greeting(name: 'John'); // Welcome to Dart John
}
```

Functions can be used as arguments to other functions. Let's see an example:

```
void welcome(element) => print('Welcome to Dart element $element');

void main() {
  var list = [1, 2, 3];
  list.forEach(welcome); // Welcome to Dart element 1
                        // Welcome to Dart element 2
                        // Welcome to Dart element 3
}
```

Note that the function signature must exactly match the expected signature of the method. For instance, you could not pass a function to the `forEach` method that expected two required arguments, or that returned a value. Let's see an example:

```
void welcome(element, position) => print('Welcome to Dart element $element');

void main() {
  var list = [1, 2, 3];
  list.forEach(welcome); // This will throw an exception and stop execution.
}
```

This following example also will throw an exception:

```
int welcome(element) => element * 2;
void main() {
  var list = [1, 2, 3];
  list.forEach(print(welcome));
}
```

To properly run the above code, it should be changed to this version:

```
int welcome(element) => element * 2;
void main() {
  var list = [1, 2, 3];
  list.forEach((i) => print(welcome(i))); // 2
                                          // 4
                                          // 6
}
```

You can also assign a function to a variable. In this particular case, our function does not have name, so it will be identified by the variable name. Thus, if we need to execute this function we will call the variable followed by parentheses and parameters if the function has them.

```
var hello = (element) => print('Welcome to Dart element $element');
void main() {
  hello(1); // Welcome to Dart element 1
}
```

Return Values

Earlier we learned that when a function does not return any value, as a rule of syntax and style we should indicate the reserved word `void`. Actually, this is not quite true, because all functions in Dart will return a value; when your function doesn't explicitly indicate a return value, the function will return `null`.

```
num sum(num a, num b) {
  return a+b;
}
void main() {
  var result = sum(1, 5);
  print(result); // 6
}
```

In the previous example, the `sum` function returns a number.

```
void greeting() {
  print('Hello!');
}

void main() {
  var result = greeting(); // Executes print Hello!
  print(result);           // null
}
```

In this case, the function executes the `print` statement, showing us the `'Hello!'` message, but it doesn't return any value, and therefore Dart will return `null`.

In the same way that you could use a function as an argument to another function, you also could use a function as a return value for other function. See the following.

```
String formatNums(num digit) {
    return digit.toStringAsPrecision(3);
}

String sum(num a, num b) {
    return formatNums(a+b);
}

void main() {
    print(sum(5, 3));           // 8.00
    print(sum(1.34578, 0.7863123)); // 2.13
}
```

In the previous sample we've shown you a function call as a return value for another function, but you can even return a function itself, as you can see below.

```
dynamic sum (num a) {
    return (b) => a + b;
}

void main() {
    var sumThree = sum(3);
    print(sumThree);    // Closure: (dynamic) => dynamic
    print(sumThree(5)); // 8

    // We can do the same as above in this way
    print(sum(3));      // Closure: (dynamic) => dynamic
    print(sum(3)(5));   // 8
}
```

Recursive Functions

As was mentioned earlier, you could have a function that calls itself. This type of function is known as a recursive function, and they're useful for showing how you could resolve a problem using only one function that executes itself repeatedly. The most important part of a recursive function is the base case, which will stop the recursion and solve the problem. If you do not properly define your base case, the recursive function will execute over and over again, similar to an infinite loop.

```
num factorial(num f) {
    if(f < 1) {
        return 1;           // base case
    } else {
        return f * factorial(f-1); // recursive call
    }
}

void main() {
    print(factorial(5));
}
```

If we haven't properly defined the base case, we could have problems with our isolates, and thus Dart will stop our execution and inform us of a `StackOverflow` error.

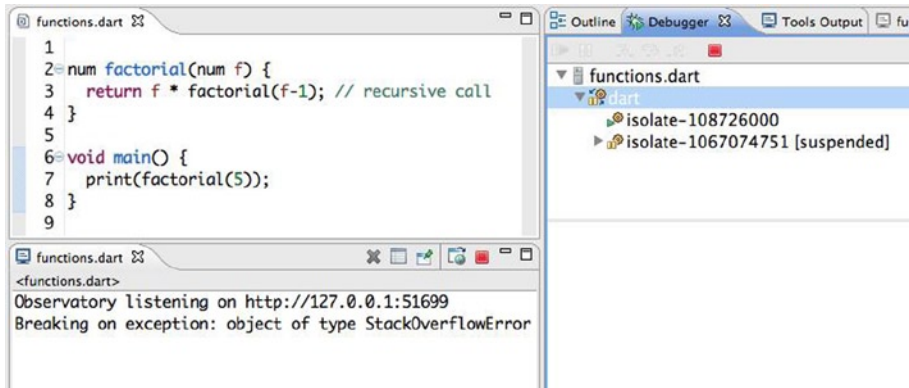


Figure 15-1. *Stack overflow error in Dart*

Summary

In this chapter we have learned the following:

- How to define functions in Dart
- What optional parameters are and which types we can use
- How to use positional parameters in functions and how to use them with default values
- How to use named parameters in functions and how to use them with default values
- What the return values are
- How to use functions as return values for other functions
- What recursive functions are and how to create recursive functions in Dart



A Syntax Summary and Cheat Sheet

In this chapter we want to give you a handy cheat sheet with the syntax learned in this book thus far (see Table 16-1 and Table 16-2). When you learn a new programming language, it is very useful to have such a thing so you can quickly check the main instructions. Here's the summary of syntax we have learned until now.

Table 16-1. *Operators cheat sheet*

Negation	[!, -]
Multiplication	[*]
Division	[/]
Integer Division	[~/]
Modulo	[%]
Addition	[+]
Subtraction	[-]
Bitwise left shift	[<<]
Bitwise right shift	[>>]
Greater than	[>]
Greater than or equal to	[>=]
Less than	[<]
Less than or equal to	[<=]
Equal to	[==]
Not equal to Sheet	[!=]

Table 16-2. *Syntax Cheat Sheet*

Inline comments	// This is an inline comment
Block comments	/* This is a block comment */
Documentation comments	/** * Doc comment * each line preceded by * * /
	/// inline doc comment
Override an operator	return_type operator name (parameters and types)
Increments expr by one and returns expr	++expr
Returns expr and increments by one its value	expr++
Decrements expr by one and returns expr	--expr
Returns expr and decrements by one its value	expr--
Assignment	[=]
Other operators to assign value	[*=], [/=], [~/=], [%=], [+=], [-=], [<=<=], [>=>=], [&=], [^=], [=]
Defines a variable	var nameOfVariable
Defines a constant	const nameOfConstant valueOfConstant
Defines a string variable	string_1 = "Example" string_1 = "Example \${other variable value} of string"
Defines string variables with multiple lines	string_1="" content of the string variable "";
Types of numbers	num, int, double
Boolean values	true, false
Creates a new List object	List l = new List() List l = [];
Defines a Map object	Map m = new Map() Map m = {};
Defines a Set object	new Set()

(continued)

Table 16-2. *(continued)*

If-else statement	<pre> if (condition) { // instructions } else { // instructions } </pre>
Switch statement	<pre> switch(variable) { case value1: // instructions break; case value2: // instructions break; default: // instructions } </pre>
While loop	<pre> while(condition) { // instructions } do { // instructions } while(condition); </pre>
For loop	<pre> for(initialization; condition; inc./dec.) { // instructions } </pre>
For-in loop	<pre> for(element in Iterable) { // instructions } </pre>
ForEach method	<pre> Iterable.forEach((e) { // instructions }); </pre>
Function definitions	<pre> return_type function_name(parameters and types) { // function body } </pre>
Simple functions	<pre> return_type function_name(parameters and types) => body_of_the_function; void greet(name) => print("Hello \$name!"); </pre>

(continued)

Table 16-2. *(continued)*

Positional optional parameters	<pre>return_type function_name([type param1, type param2, ...]) { // function body }</pre>
Named optional parameters	<pre>return_type function_name({type named_param1, type nane_param2, ...}) { // function body }</pre>
Default function values	<pre>return_type function_name([type param1=value, type param2=value, ...]) { // function body }</pre> <pre>return_type function_name({type param1: value, type param2: value, ...}) { // function body }</pre>

PART IV



The Dart Language: Advanced



Processing Exceptions and Error Control

Thus far we've looked at important aspects of Dart, at what we consider to be the first part of the language. Now we're going to dive in a little deeper. In this second part of the book we'll learn about error control and exceptions—everything related to the creation and use of classes, constructors, and inheritance.

We'll also talk about generics, isolates, typedefs, and how libraries work in Dart. Do not be scared by these weird names, as you will soon see what they are and how you can use them to improve your knowledge of Dart.

Exceptions and Error Control

All modern and object-oriented languages, as Dart is, should include exceptions and error control. When you are programming an application, in spite of the fact that you've tried to exert as much control as possible, there will always be a set of data that you have not tried or a user behavior on a certain screen that won't be something you defined for. In these cases, there will be errors, and your application will fail or will be blocked.

Dart incorporates an exceptions mechanism to capture exceptional error cases not previously considered, and it'll act accordingly. As a result, you can display a more appropriate error message for the user and keep the system running.

In other cases, you can throw exceptions within specific blocks of your application. Dart also incorporates mechanisms to do this. When something unusual happens in your application, Dart throws an exception, and if the exception is not caught, the isolate that generated the exception will be interrupted and the program will end.

Dart provides `Exception` and `Error` for managing errors and exceptions. It also lets you create your own exceptions. Dart's mechanisms to manage errors and exceptions are called `throw`, `catch`, and `finally`. You may know these methods because they're similar to those found in PHP.

`Throw` lets you hurl exceptions anywhere in your code. Let's see a very simple example:

```
if(user == null) {  
  throw new Exception('User not recognized in the system !!');  
}  
  
if(password == null) {  
  throw new Exception('The password is incorrect !!');  
}  
  
if(data == null) {  
  throw new ArgumentError();  
}
```

```
if(data is! Map) {
  throw new TypeError();
}
```

Catch is Dart's mechanism to capture and handle exceptions, thus the propagation of the exception stops and you can manage it. To be more exact, Dart catches exceptions using the `try {} on {} catch {}` blocks. These blocks can be nested and thus can capture more than one type of exception. See here:

```
try {
  login();
} on PasswordEmptyException { // Catches a specific exception.
  reLogin();
} on Exception catch(e) { // Catches any type of exception.
  print('What happend here?');
} catch(e) { // Catches anything else than will happen.
  print('Something has gone wrong. I have no idea what happened !');
}
```

As you can see from this example, you can chain together try-catch blocks, depending on the types of exceptions you want to capture and handle.

During the capture of an exception you can use `on`, `catch`, or both. Use `on` when you need to specify the exception type. Use `catch` when your exception handler needs the exception object.

Finally, Dart gives you an opportunity to execute a code snippet at the end of the try-catch block, whether the exception occurs or not. With `finally`, you will always execute the code snippet, as follows:

```
var myresult;
try {
  // I try to gain access to the system.
  myresult = login();
} on Exception catch(e) {
  // An error occurs in the system.
  myresult = 'You did not say the magic word !';
} finally {
  // Displays the answer to the user.
  print(myresult);
}
```

You can find the `Exception` class and the `Error` class on `dart:core`. Use them to manage exceptions or errors in the system; these classes also allow you to define your own exceptions and errors.

Exceptions Types

There are three types of exceptions in Dart within the exceptions package on `dart:core`, as follows:

- `Exception` is the abstract class that represents a system failure. You must inherit from `Exception` class in order to create your own exceptions.
- `FormatException` defines a format error when a string or other value does not represent the correct format.
- `IntegerDivisionByZeroException` represents a division by zero exception.

Error Types

In the errors package on `dart:core` there are some classes that define different types of common errors:

- `AssertionError` is used when an `assert` statement fails.
- `TypeError` is used when a type `assert` statement fails.
- `CastError` is used when trying to convert a value from one type to another and failure occurs.
- `NullThrownError` is used when you throw `null` on your code.
- `ArgumentError` will occur when an error occurs in function arguments.
- `RangeError` is used when you try to access an index that is outside the range of indexes of the object.
- `FallThroughError` is used when control reaches the end of a switch case.
- `AbstractClassInstantiationError` is used when trying to instantiate an abstract class.
- `NoSuchMethodError` is used when you try to access a nonexistent method on an object.
- `UnsupportedError` is used when an operation is not allowed by an object.
- `UnimplementedError` is used when the method is not implemented in the object.
- `StateError` will throw when the operation was not allowed by the current state of the object.
- `ConcurrentModificationError` occurs when a collection is modified during iteration.
- `OutOfMemoryError` is used when the system is out of memory.
- `StackOverflowError` is used when a stack overflow failure occurs.
- `CyclicInitializationError` will be thrown when a lazily initialized variable cannot be initialized.

Exceptions and Error Definitions

Creating exceptions and errors in Dart is really simple—you just need to create the class of exception or error that you want to define and then extend the basic Dart error or exception classes.

What's the difference between an error and an exception? In particular, errors are meant to be fatal and halt program execution (and will not be caught in most cases). Exceptions are designed to be used for cases in which an error is expected, such as parsing user input (`FormatException`).

Let's look at some simple examples of defining exceptions and errors and how to use them. To define your own exception class you must implement the base class `Exception`:

```
class MyException implements Exception {
  final msg;
  const MyException([this.msg = ""]);
  toString() => 'Fatal Exception in the system.$msg';
}
```

And then you can use it on your system in this way:

```
throw new MyException(); // Exception: Fatal Exception in the system.
throw new MyException('Access Error.'); // Exception: Fatal Exception in the system.Access Error
```

To define your own error classes you must extend the base class `Error` as follows:

```
class MyError extends Error {
  final msg;
  MyError([this.msg = " UNKNOWN "]);
  toString() => 'Error: $msg';
}
```

Then use your new error class as follows:

```
throw new MyError(); // Exception: Error: UNKNOWN
throw new MyError('Access Denied.'); // Exception: Error: Denied Access.
```

We know we haven't previously discussed classes, yet in this chapter we have talked about Exception and Error classes and used **extends** and **implements** keywords—do not worry about that, as we'll see these items in the next chapter.

Now let's get started talking about classes and explaining the difference between **extends** and **implements**!

Summary

In this chapter we have learned:

- How to control errors and manage exceptions
- When you should use `on` or `catch` to handle errors and exceptions
- What types of exceptions there are in Dart
- What types of errors there are in Dart
- How to create custom error classes
- How to create custom exception classes



Understanding Dart Classes

In this chapter we will cover Dart classes, what constructors are, and how to create constructors for your classes. We'll see named constructors, instance variables, and instance methods as well. Later we'll see how inheritance works in Dart, and we'll talk about static variables and methods.

As you know, Dart is an object-oriented language with classes and single inheritance. Everything in Dart is an object, including the built-in data types. An object is an instance of a class. To create an object, use the keyword `new` followed by the constructor of the class. See the following:

```
class Person {  
  Person() {  
    // Do stuff here  
  }  
}  
  
void main() {  
  Person john = new Person(); // John is an object, an instance of Person.  
}
```

Note that we can even use the `Person` class as a type annotation of our new variables.

Objects have members, functions, and data. The methods of a class are the functions, and instance variables are the data.

```
class Person {  
  // Instance variables.  
  String name;  
  num age;  
  
  // Empty constructor.  
  Person();  
  
  // Method that displays a greeting.  
  void sayHello() => print('Hi my name is $name');  
}  
  
void main() {  
  Person john = new Person(); // John is an object, the instance of Person.  
  john.name = 'John Doe';  
  john.age = 29;  
  john.sayHello(); // Hi my name is John Doe  
}
```

■ **Note** The style guide recommends using a semicolon rather than empty braces for empty constructors.

In this example we created a simple class with some instance variables and one method in addition to the constructor. Once you define the object, you can access its instance variables or methods using the object name followed by a dot and the method or instance variable you want to get.

Just as with variables, you can define objects as being either `const` or `final`, and with classes you can do the same thing. You can define a class that has a constant constructor so that a constant object is created at runtime. It creates the object by using the word `const` instead of `new` when calling the constructor of that class.

```
class DaVinci {
  final name = 'Leonardo da Vinci';
  const DaVinci(); // Constant Constructor.
}

void main() {
  var leo = const DaVinci();
  print(leo.name); // Leonardo da Vinci
}
```

Instance Variables

We have previously commented that the instance variables are the “data” of the object. These instance variables are defined in the class and are declared like variables, as we have already learned.

We indicate (optionally) the type, the name, and an assigned value; if you don’t indicate a value, the default value will be `null`.

All instance variables defined will generate an implicit getter method (to get the value of the instance variable), so you will not have to declare it. Dart simplifies things a lot.

All instance variables defined with a data type of `var` also will generate an implicit setter method (to set the value of the instance variable).

We have seen in the above example the `Person` class and how we could set the value of the name and the age, thus displaying them without creating these getter or setter methods.

```
class Person {
  // Instance variables.
  String name;    // Variable initialized to null.
  num age;        // Variable initialized to null
  num childs = 0; // Variable initialized to 0.

  // Constructor.
  Person();

  // Method that displays a greeting.
  void sayHello() => print('Hi my name is $name');
}
```



```

void main() {
  Person john = new Person(); // John is an object, an instance of Person.

  // we can set the name and the age without having to define setter methods.
  john.name = 'John Doe'; // we set the name
  john.age = 29;           // we set the age

  // we can display the name and the age without having to define getter methods.
  print('Name: ${john.name}'); // Name: John Doe
  print('Age: ${john.age}');   // Old: 29
}

```

Instance variables can be visible or not visible from other libraries, which means you have the option of accessing them from the instance created or not. The instance variables that are not visible are called *private*.

If you start the variable name with an underscore (`_`) you are indicating that this instance variable is private.

```

class Person {
  // Instance variables.
  String name;
  num age;
  num _id; // Private variable

  // Constructor.
  Person();

  void sayHello() => print('Hi my name is $name');
}

```

You can identify private variables in the Outline by the red icon assigned to private instance members, as shown in Figure 18-1.

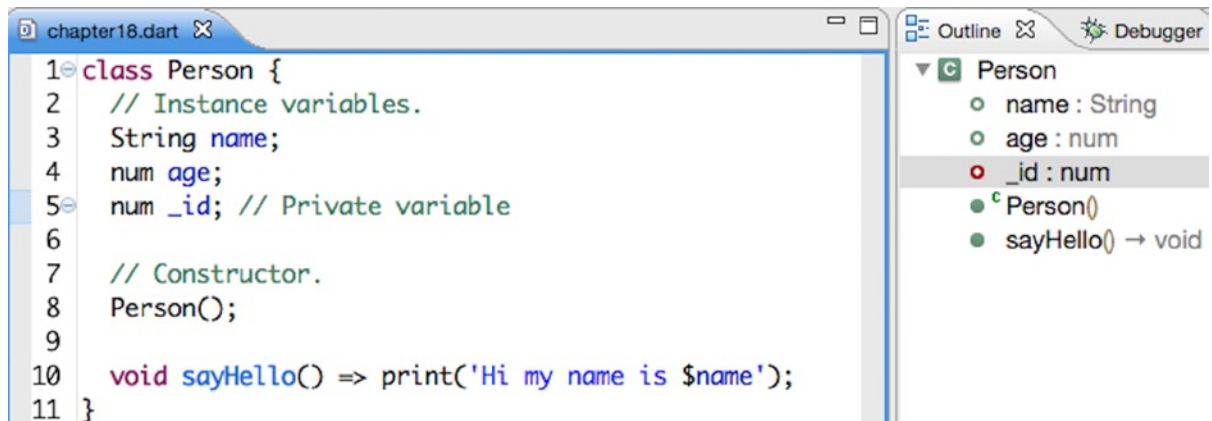


Figure 18-1. Dart editor outline displaying private/public instance variables

Constructors

In our previous examples we have also seen something about class constructors. The constructor of the class is a class method that is executed when the object is created and the class is instantiated. The constructor is a special method that has the same name as the class. Here's an example:

```
class Person {
  // Instance Variables
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num children = 0; // the variable is initialized to 0.

  // Constructor.
  Person(String name, num age) {
    this.name = name;
    this.age = age;
  }

  // Method launching a greeting.
  void sayHello() => print('Hi my name is $name and i\'m $age years old.');
```

```
}

void main() {
  Person p = new Person('John Doe', 26);
  p.sayHello(); // Hi my name is John Doe and i'm 26 years old
}
```

Inside the constructor we can refer to instance variables using the keyword `this`. If you have worked with JavaScript, Java, or C# before, this will be familiar for you.

The Dart style guide discourages using `this` regularly and recommends it for use only when you have a name conflict. Here you can see the same example without using the `this` keyword inside our constructor; obviously we have to solve the name collision as well.

```
class Person {
  // Instance Variables
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num chidren = 0; // the variable is initialized to 0.

  // Constructor.
  Person(String n, num a) {
    name = n;
    age = a;
  }

  // Method launching a greeting.
  void sayHello() => print('Hi my name is $name and i\'m $age years old ');
}
```

In the same way as you define instance variables with its type annotation or with the `var` keyword, if you do not specify a constructor for the class, Dart implicitly creates one without arguments, and this will be the first method to be executed when the object is created.

Within constructors you should pay attention to named constructors, constructors with names that define multiple constructors for a class.

Named constructors are not new to you, as you saw them earlier when we covered the `DateTime` class. As you may recall, there are different ways of defining a `DateTime` object: `DateTime.now()`, `DateTime.utc()`, or `DateTime.fromMillisecondsSinceEpoch()`.

Named constructors can be used for different reasons—to initialize instance variables or to clarify the functions of the class.

```
class Person {
  // Instance Variables
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num children = 0; // the variable is initialized to 0.

  // Constructor.
  Person(String n, num a) {
    name = n;
    age = a;
  }

  // Named constructor.
  Person.json(Map data) {
    name = data['name'];
    age = data['age'];
    children = data['children'];
  }

  // Method launching a greeting.
  void sayHello() => print('Hi my name is $name and i\'m $age years old.');
```

```
}

void main() {
  Person p = new Person('John Doe', 26);
  p.sayHello(); // Hi my name is John Doe and I'm 26 years old

  var p2 = new Person.json({
    'name': 'Michael',
    'old': 45,
    'children': 2
  });
  p2.sayHello(); // Hi my name is Michael and I'm 45 years old.
}
```

Let's talk now about *constructor shortcuts*. These shortcuts allow us to save on typing when we're writing our Dart classes. Check this sample out:

```
class Person {
  String name;
  num age;

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

We can use constructor shortcuts to avoid repetition when the constructor argument's name is the same as the name of the field. Let's see now how our sample looks using the constructor shortcuts:

```
class Person {
  String name;
  num age;

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

We've used the `this` keyword on the constructor parameters. Note that the `this` keyword refers to the current instance. The Dart style guide usually omits the `this` keyword, and you must use it only when there is a name conflict.

Let's see now another great feature you can use on class constructors, the *initializer list*. In some cases you can also initialize instance variables before the constructor body runs. You can separate the initializers with commas, and you can use the constructors' parameters to initialize instance variables. See here:

```
class User {
  String name;
  String password;
  bool loggedin = false;

  User.fromJSON(Map json)
    : name = json['name'],
      loggedin = true,
      password = json['password'] {
    print('User initialized');
  }
}
```

When you define a constructor, you can redirect it, which is especially useful when there is an inheritance from the constructor of the child class. In this case we can redirect to the constructor of the parent class. You can also do it within the class itself if the utility of the class is clarified.

```
class Person {
  //Instance Variables.
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num children = 0; // the variable is initialized to 0.
```

```
// Constructor.
Person(String n, num a) {
  name = n;
  age = a;
}

Person.withoutChildren(String n, num a):this(n, a);

// Method launching a greeting.
void sayHello() => print('Hi my name is $name and I\'m $age years old.');
```

```
}

void main() {
  var p = new Person.withoutChildren('Alice', 19);
  p.sayHello(); // Hi my name is Alice and I'm 19 years old.
}
```

As we discussed earlier, there are also constant constructors. You must use these if you know that an object does not change throughout the execution of the application. Create the class with its constant constructor, using `const`, and make sure all instance variables are constant with either `final` or `const`.

```
class Origin {
  final num x;
  final num y;
  const Origin(this.x, this.y);
}
```

Finally, let's talk about the factory constructors. When you use factory constructors to instantiate a class, a new instance won't be created; instead, the object is retrieved from a cache, so this can save a lot of memory in your applications.

The factory constructors are usually used when you are interested in separating the creation of an object from its implementation, and they are very useful when you need to choose between a number of interchangeable classes at runtime.

In other languages, the factory pattern is also used, but you must use a combination of static methods, class utilities, and so forth. Dart has already included this pattern, and you can use it natively. Here's an example:

```
class Boss {
  // Boss name
  String name;

  // cache to store the created instance.
  static final Map<String, Boss> _cache = <String, Boss>{};

  Boss._internal(this.name);

  // This is our factory constructor.
  factory Boss(String name) {
    if(_cache.containsKey(name)) {
      return _cache[name];
    }
  }
}
```

```

    } else {
      final boss = new Boss._internal(name);
      _cache[name] = boss;
      return boss;
    }
  }

  void sayHello() => print('Hi I\'m $name, the Boss!!');
}

void main() {
  var j1 = new Boss('Bruce Springsteen');
  var j2 = new Boss ('Bruce Springsteen');

  j1.sayHello(); // Hi I'm Bruce Springsteen, the Boss!!
  j2.sayHello(); // Hi I'm Bruce Springsteen, the Boss!!
  print(j1==j2); // true.
}

```

Note that j1 and j2 are in fact the same object.

Methods

Throughout the previous examples we have seen various class methods. Methods endow functionality to your objects. It is a function defined inside a class that ensures that the object has useful features.

```

class Person {
  // Instance Variables
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num children = 0; // the variable is initialized to 0.

  // Constructor.
  Person(this.name, this.age);

  void sayHello() => print('Hi my name is $name and I\'m $age years old.');
```

```

  void work() => print('Working ..');
  void eat() => print('Ummm delicious !');
  void sleep() => print('Good Night.. ZZZZZZZZ');
  void getSomeFun() => print('I love play HayDay');
```

```

}

void main() {
  var p1 = new Person('Mary', 19);
  p1.sayHello(); // Hi my name is Mary and I'm 19 years old.
  p1.eat();      // Ummm delicious !
  p1.work ();    // Working...
  p1.eat();      // Ummm delicious !
  p1.getSomeFun(); // I love play HayDay
  p1.sleep();    // Good Night.. ZZZZZZZZ
}

```

In addition to instance methods for your objects, there are also getters and setters, which we previously discussed and which Dart will create automatically. However, if you needed to create a getter or a setter for any of your classes, you can do so. See here:

```
class Person {
  // Instance Variables
  String name;      // the variable is initialized to null.
  num age;          // the variable is initialized to null.
  num children = 0; // the variable is initialized to 0.

  Person(this.name, this.age);

  bool get isParent => (children > 0);
  set parent(bool p) => (p==true)?children=1:children=0;
}

void main() {
  var p1 = new Person('Mary', 19);
  print(p1.isParent); // false

  p1.parent = true;
  print(p1.isParent); // true
  print(p1.children); // 1
}
```

When you define your getter and setter methods, you should use the abbreviated form as recommended in the Dart style guide; you must define type annotations for the return value in the getter methods and for the parameters in the setter methods.

When you define a class you can also those override operators that you learned in the first chapters, for example, +, -, and so forth.

Imagine you have a class that defines a GPS position. You may be interested in defining the operators + and - if you want to add or subtract two GPS positions and get a third GPS position as a result. Here's an example of how to do that:

```
class GPS {
  num latitude;
  num longitude;

  GPS(this.latitude, this.longitude);

  GPS operator +(GPS g) {
    return new GPS(latitude + g.latitude, longitude + g.longitude);
  }

  GPS operator -(GPS g) {
    return new GPS(latitude - g.latitude, longitude - g.longitude);
  }
}
```

```
void main() {
  var g1 = new GPS(40.311235, 32.87578786);
  var g2 = new GPS(38.642331, 41.79786401);
  var g3 = g1 + g2;
}
```

The Operators you can override in your classes are shown in Table 18-1.

Table 18-1. *Override Operators*

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

Inheritance

Inheritance allows you to create new classes from existing ones by adding a new behavior, more information, or more functionality to classes that inherit from parent classes, thus avoiding the redesign, modification, and verification of code already implemented.

In Dart you can create subclasses using the word `extends`, and you can refer to the parent class with the word `super`:

```
// Base class for animals
class Animal {
  String species;
  String variety;

  Animal(this.species, this.variety);

  void breathe() {
    print('Configuring respiratory system...');
    print('Respiratory system configured!');
  }

  void eat() {
    print('Configuring digestive system...');
    print('Digestive system configured !');
  }
}

// Concrete class for Dogs
class Dog extends Animal {
  String colour;
  num paws;
```



```

Dog(String variety, String color, num paws) : super('mammal', variety) {
  this.colour = colour;
  this.paws = paws;
}

void breathe() {
  super.breathe();
  print('I\'m breathing by muzzle');
}

void main() {
  var p = new Dog('Golden Retriever', 'yellow', 4);
  p.breathe(); // Configuring respiratory system...
               // Respiratory system configured !
               // I'm breathing by muzzle
  p.eat();     // Configuring digestive system...
               // Digestive system configured !
}

```

In the child class you can create new instance variables or new methods; you can also rewrite the parent methods. Whenever you need to refer to a method or instance variable of the parent class, use the word `super`.

In talking about inheritance we must say that constructors are not inherited, and neither are named constructors.

A subclass does not inherit the constructors of the parent class. If a constructor is not defined to the new subclass, then it will have the implicit constructor that Dart creates, without parameters:

```

class Animal {
  String name;
  String type;
  String breed;
  num age;
  num paws;

  Animal();

  Animal.json(Map data) {
    name = data ['name'];
    type = data ['type'];
    breed = data ['breed'];
    age = data ['age'];
    paws = data ['paws'];
  }
}

class Dog extends Animal {
  Dog.json(Map data) : super.json(data);
}

```

■ **Note** To execute a parent constructor in the subclass, you must do it explicitly. You must define the constructor in the child class and then inside the constructor use the word `super` to call the constructor parent.

Variables and Static Methods

In Dart it is possible to create static methods and static variables when you define your classes. Static means you do not need to instantiate the class to use them, and they have a local scope to the class where they are defined, but their existence is permanent.

To create static methods or variables it is only necessary to use the word `static`.

```
class Car {  
  static num maximum_speed = 120;  
}  
  
void main() {  
  print(Car.maximum_speed); // 120  
}
```

The static methods do not operate on the instances, so they cannot use the reserver keyword `this`.

```
class _CryptoUtils {  
  
  static String bytesToHex(List<int> bytes) {  
    var result = new StringBuffer();  
    for (var part in bytes) {  
      result.write('${part < 16 ? '0' : ''}${part.toRadixString(16)}');  
    }  
    return result.toString();  
  }  
}  
  
void main() {  
  print(_CryptoUtils.bytesToHex([6,4,7,8,1,0])); // 060407080100  
}
```

■ **Tip** The Dart style guide recommends using global functions instead of classes with static methods for tasks or utilities commonly used in your applications.

Summary

In this chapter we have learned:

- How to create classes in Dart
- What the instance variables and the instance methods are
- The different types of constructors we can use in Dart
- About constructor shortcuts and initializer lists
- How to create subclasses from a parent class and how to call methods or constructors on the parent class
- What static variables and methods are and how to define them in your classes



Implicit Interfaces and Abstract Classes

In this chapter we'll cover interfaces and abstract classes in Dart. We'll also see the difference between implementing interfaces and extending from a parent class. When you define a class, Dart implicitly defines an interface that contains all members of the instance and methods of the class and of the rest of the interfaces that implements. In short, if you want to write a class *A* that supports the API of a class *B* without inheriting the implementation of *B*, the *A* class must implement the *B* interface. To do this, when you define the class you can add the word `implements` followed by the classes you want implement and define the interfaces implementation.

Let's see an example to clarify this. We'll create a class and its implicit interface. Then we'll create another class which will implement the first one.

```
// In the implicit interface of Animal class is located the speak() method
class Animal {
  String _name;      // It is present in the interface
// but it's only visible in Animal class.
  Animal(this._name); // The constructor is not present in the interface.
  void speak() => print('Hi I\'m an Animal and my name is $_name');
}

// Implementation of the Animal interface.
class Dog implements Animal {
  String _name; // It should be defined because
                // it was only visible in the Animal class.
  void speak() => print('Hi I\'m a Dog and I have no name');
}

// Function that receives an object and executes its speak() method
sayHello(Animal animal) => animal.speak();

void main() {
  sayHello(new Animal('Boby')); // Hi I'm an Animal and my name is Boby
  sayHello(new Dog());          // Hi I'm a Dog and I have no name.
}
```

In this simple example, we've implemented the `Animal` interface in the class `Dog`. We must implement everything in the implicit interface from the `Animal` class methods and instance variables. We implemented the `speak()` method and the instance variable `_name`, although later it is not used. If you look at the Dart SDK, you can see how this mechanism is very common. Figure 19-1 provides an example of implementing multiple interfaces, available in `dart.html`.

```
@DocsEditable()
/**
 * An abstract class, which all HTML elements extend.
 */
@DomName('Element')
abstract class Element extends Node implements GlobalEventHandlers, ParentNode, ChildNode {

  /**
   * Creates an HTML element from a valid fragment of HTML.
   *
   * var element = new Element.html('<div class="foo">content</div>');
   *
   * The HTML fragment should contain only one single root element, any
   * leading or trailing text nodes will be removed.
   *
   * The HTML fragment is parsed as if it occurred within the context of a
   * `<body>` tag, this means that special elements such as `<caption>` which
   * must be parsed within the scope of a `<table>` element will be dropped. Use
   * [createFragment] to parse contextual HTML fragments.
   *
   * Unless a validator is provided this will perform the default validation
   * and remove all scriptable elements and attributes.
   *
   * See also:
   *
   * * [NodeValidator]
   */
  factory Element.html(String html,
    {NodeValidator validator, NodeTreeSanitizer treeSanitizer}) {
    var fragment = document.body.createFragment(html, validator: validator,
      treeSanitizer: treeSanitizer);

    return fragment.nodes.where((e) => e is Element).single;
  }
}
```

Figure 19-1. The `Element` class inherits from `Node` and implements the interfaces of `GlobalEventHandlers`, `ParentNode` and `ChildNode`

In the previous example you can see how `Element` extends from `Node` and implements the interfaces of `GlobalEventHandlers`, `ParentNode`, and `ChildNode`. `Implements` allows your classes to inherit from multiple classes.

So, what's the difference between `extends` and `implements`? As we've seen in Chapter 18 the keyword `extends` is used when you want to create subclasses from a parent class inheriting instance members from the parent class.

The subclass will have all the methods and instance variables located on the parent class, but you don't need to define them again as you must do with `implements`. Let's see a simple example of `extends` and `implements`.

```
class Animal {
    String name;
    Animal(this.name);
    void greet() => print('Hi!');
}

class Dog extends Animal {
    Dog(String name):super(name);
    void eat() => print('eating...!');
}
```

The `Dog` class can use `name` and `greet()` and it doesn't need to define them again.

```
class Animal {
    String name;
    Animal(this.name);
    void greet() => print('Hi!');
    void eat() => print('eating...');
}

class Dog implements Animal {
    String name;
    Dog(this.name);
    void greet() => print('Woof!');
    void eat() => print('ummmm...');
}
```

In this case the `Dog` class must define the `name`, `greet()`, and `eat()` methods which are present in the interface. `Extends` is mainly used to reduce codebase by factoring common code into a super class. Interfaces are used to describe functionalities that are shared by a number of classes. With any discussion of interfaces, you must know what *abstract classes* are. An abstract class is defined as a regular class, but you must use the keyword `abstract` before the class definition

WHAT DOES ABSTRACT CLASS MEAN?

When we do a class hierarchy, sometimes there is a behavior that will be present in all the classes but it materializes differently for each one. For these purposes, we use abstract classes.

The abstract classes defining a common behavior to all classes will inherit or implement this class. Abstract classes are also very useful for defining interfaces. Not only can the classes be abstract, *methods* also can be *abstract*. The process is the same, an abstract method will be a common function present in all classes, but it will work differently in each different class. When you're going to define your abstract classes or abstract methods, usually they will be defined without implementation and classes which implement this abstract class will define the code. Only the class is defined using the `abstract` keyword. Abstract methods can only be defined implicitly by omitting their bodies.

// Defines GeometricFigure interface. It won't be never instantiated.

```
abstract class GeometricFigure {
    // Define constructors, methods and instance variables.
    void draw(); // Abstract method.
}
```

```
class Circle extends GeometricFigure {
    void draw() {
        // Put your code here to draw a circle.
    }
}
```

We're going to see another example of abstract classes and how to use them. We're going to create two abstract classes to represent musical instruments and stringed instruments and how other classes, such as `ElectricGuitar`, `SpanishGuitar`, and `BassGuitar` can use these abstract classes.

```
// Abstract class for all the musical instruments.
abstract class Instrument {
    String name;
    void play();
}
```

```
// Abstract class for all the stringed instruments.
abstract class StringedInstrument extends Instrument {
    int number_of_strings;
}
```

```
// Concrete class for electric guitars.
class ElectricGuitar extends StringedInstrument {
    String name;
    int number_of_strings;

    ElectricGuitar(this.name, this.number_of_strings);

    void play() {
        print('$name Electric Guitar with $number_of_strings strings playing rock !');
    }
}
```

```
// Concrete class for Spanish guitars
class SpanishGuitar extends StringedInstrument {
    String name;
    int number_of_strings;

    SpanishGuitar(this.name, this.number_of_strings);

    void play() {
        print('$name Spanish Guitar with $number_of_strings strings playing flamenco !');
    }
}
```

```
// Concrete class for bass
class BassGuitar extends StringedInstrument {
  String name;
  int number_of_strings;

  BassGuitar(this.name, this.number_of_strings);

  void play() {
    print('$name Bass Guitar with $number_of_strings strings playing Jazz !');
  }
}

void main() {
  ElectricGuitar eg = new ElectricGuitar('Fender', 6);
  eg.play(); // Fender Electric Guitar with 6 strings playing rock !

  SpanishGuitar sg = new SpanishGuitar('Manuel Rodriguez', 6);
  sg.play(); // Manuel Rodriguez Spanish Guitar with 6 strings playing flamenco !

  BassGuitar bg = new BassGuitar('Ibanez', 4);
  bg.play(); // Ibanez Bass Guitar with 4 strings playing Jazz !
}
```

We've talked about implicit interfaces and abstract classes. Now it's time to talk about Mixins. *Mixins* is a way of reusing a class's code in multiple class hierarchies.

You can use mixins to inject functionalities into your classes without using inheritance.

To define a mixin in Dart you can use an abstract class with a few restrictions.

- Create a class that extends Object. In Dart all the classes extend from Object by default.
- The class has no declared constructors.
- The class contains no super calls.

According to our previous example where we define musical instruments we can change our `StringedInstrument` abstract class and use them as a mixin for the guitar instrument classes. Let's see how to define and use mixins.

```
// Instrument extends from Object
// Has no declared constructors
// Has no calls to super
// It's a Mixin!
abstract class Instrument {
  String name;
  void play() => print('Playing $name ... ');
}

// StringedInstrument extends from Object
// Has no declared constructors
// Has no calls to super
// It's a Mixin!
```



```

abstract class StringedInstrument {
    int number_of_strings;
    void tune() => print('Tunning $number_of_strings strings instrument ... ');
}

// Concrete class for electric guitars.
class ElectricGuitar extends Object with Instrument, StringedInstrument {
    String name;
    int number_of_strings;

    ElectricGuitar(this.name, this.number_of_strings);
}

void main() {
    ElectricGuitar eg = new ElectricGuitar('Fender', 6);
    eg.tune(); // Tunning 6 strings instrument ...
    eg.play(); // Playing Fender ...

    print(eg is ElectricGuitar);    // true
    print(eg is StringedInstrument); // true
    print(eg is Instrument);        // true
}

```

As you can see in this example you can use `play()` and `tune()` methods in the `ElectricGuitar` object without inheritance because of the mixins. Notice that a class that uses a mixin is also a member of the mixin's type.

We changed our instruments example to use mixins in order to show you how you can modify your classes from inheritance to mixins. But this is not the best example to perfect understand mixins.

Generally you should use mixins when your classes don't fit well in a *is-a* relationship. For this reason we said the above code is not the perfect example for mixins because `ElectricGuitar` is an `Instrument` so it fits in a *is-a* relationship.

Let's another example to use mixins for a class that not fit in a *is-a* relationship. Imagine you have an application to manage products. You have different types of products, for example, discs and books. Your classes look like this.

```

class Product {
    String name;
    double price;
    Product(this.name, this.price);
}

class Disc extends Product {
    String artist;
    String title;
    Disc(name, price, this.artist, this.title):super(name, price);
}

class Book extends Product {
    String isbn;
    String author;
    String title;
    Book(name, price, this.isbn, this.author, this.title):super(name, price);
}

```

```

void main() {
    var d = new Disc('Planes, trains and Eric', 19.95, 'Eric Clapton',
        'Planes, trains and Eric');
    var b = new Book('Web Programming with Dart', 31.99, '978-1-484205-57-0',
        'Moises Belchin & Patricia Juberias', 'Web Programming with Dart');
}

```

You have a new backend and now you have to send information about your products in a JSON format to the backend. So you write a class called `JSONSupport` for converting `Product` objects into JSON and send to the backend and viceversa.

None of your products fit well in a is-a relationship with your new `JSONSupport` class. So you can use Mixins to add this new functionality into your classes without change your class hierarchy. This is our new code.

```

abstract class JSONSupport {
    convertToJSON() => print('Converting to JSON object ...');
    sendToBackend() => print('Sending to Backend ...');
}

class Product extends Object with JSONSupport {
    String name;
    double price;
    Product(this.name, this.price);
}

class Disc extends Product {
    String artist;
    String title;
    Disc(name, price, this.artist, this.title):super(name, price);
}

class Book extends Product {
    String isbn;
    String author;
    String title;
    Book(name, price, this.isbn, this.author, this.title):super(name, price);
}

void main() {
    var d = new Disc('Planes, trains and Eric', 19.95, 'Eric Clapton',
        'Planes, trains and Eric');
    var b = new Book('Web Programming with Dart', 31.99, '978-1-484205-57-0',
        'Moises Belchin & Patricia Juberias', 'Web Programming with Dart');
    d.convertToJSON();
    b.sendToBackend();
}

```

As you can see now you can use `.convertToJSON()` and `.sendToBackend()` methods on your `Disc` and `Book` objects without change these classes thanks to Mixins.

Summary

In this chapter we have learned:

- What implicit interfaces are and how to use them.
- How to implement an interface.
- The difference between extends and implements.
- What abstract classes are and how to create and use them in Dart.
- What mixins are and how you can use them in Dart by adding new functionalities into your classes without inheritance.



Implementing Generics and Typedefs

As we've mentioned several times throughout this book, Dart is an optional-typed language, which means that you are not required to specify variables' data types in your applications. However, data typing can be very useful for self-documenting your source code and can make your life easier if you have to maintain the application.

If you indicate data types in your applications, something amazing will happen—all the Dart tools will begin to work for you and will provide various alerts while you work. These alerts might indicate typing errors between the parameters of a function or between a function result and assignment to a variable. Providing definitive data types in a language like Dart is like making a note to yourself, *"Hey, remember that this is a number!"*

They can also called type annotations, which are not to be confused with metadata annotations. Metadata consists of a series of annotations, each of which starts with @, followed a constant expression that starts with an identifier. This means you can use a class with a constant constructor as annotation. Metadata annotations are used to give additional information about your code.

In Dart there are three annotations available to all Dart code: @deprecated, @override, and @proxy. Let's see an example:

```
class Person{

  @deprecated // makes Dart Editor warn about using sayHello().
  void sayHello(String msg) => print(msg);

  void greet(String new_msg) => print(new_msg);
}
```

You can create your own metadata annotations, which can be as simple as creating a class with a constant constructor as you can see below. Let's create a class called FixMe to add annotations to within our code in order to fix bugs or create a better implementation. See here:

```
class FixMe {
  final String note;
  const FixMe(this.note);
}

class Person{
  @FixMe("Find better name")
  void eat() => print('eating...');
}
```

Generics

The **generics** are annotations for collections. When we covered maps and lists you saw how you can indicate the data type of the elements of that map or list. That is what the generics are.

Using angle brackets followed by the data type (`<...>`) we indicate to Dart that our list or our map will contain elements of that type. There are two different ways to make type annotations for collections:

```
List<int> l_num_1 = [1, 2, 3, 4, 5];
```

```
var l_num_2 = <int>[1, 2, 3, 4, 5];
```

What's the difference between them? The first indicates that the variable `l_num_1` is a `List<int>`, whereas the second indicates that the variable is any type that at the moment contains a list of `ints`. When adding a string to the list, the first will create a warning while the second will create only a hint. If assigning a string to the variable (replacing the list), the first will create a warning while the second will not generate a warning or hint.

Generics will let Dart easily verify the operations you perform with the elements of the collections. Dart will verify whether you are adding elements that are `nums` or `Strings` and will warn you if you're running an invalid operation over those objects.

The generics also help you to reduce the amount of code. Imagine you create a class that manipulates a data structure. You decide to define the data type of this structure to take advantage of data-type checks and warnings during development. You decide to establish types, and you're going to use `String`. Later, you realize that you've written an awesome class. This class could also be used to manipulate other data structures with `nums`. You should duplicate the class and change only the data type, right?

If you use generics there's no need to duplicate the class you've created. You can use a **generic** as a parameter in your class. With that generic parameter, you're telling Dart that this class could manage any data type. Let's see a very simple example to better understand this. We're going to create a cache class to manage the cache on the client side. This cache class is amazing, and we would like to use it with any data type. To do this we'll use `generic` as a parameter for our cache class:

```
// We define the class and indicate that parameter would be any data type.
```

```
class Cache<T> {
  Map <String, T> _cache;

  // getByKey will return any data type value.
  T getByKey(String key) {
    if(_cache.containsKey(key)) {
      return _cache[key];
    }
    return null;
  }

  // setByKey will store any data type value.
  void setByKey(String key, T value) {
    _cache[key] = value;
  }

  Cache() {
    if(_cache == null) {
      _cache = {};
    }
  }
}
```

```

void main() {
  var strings = new Cache<String>();
  strings.setByKey('one', '1');
  print(strings.getByKey('one')); // '1'

  // This statement will throw an exception
  // type 'int' is not a subtype of type 'String' of 'value'.
  strings.setByKey('two', 2);

  var numbers = new Cache<num>();
  numbers.setByKey('two', 2);
  print(numbers.getByKey('two')); // 2
  print(numbers.getByKey('one')); // null
  // This statement will throw an exception
  // type 'String' is not a subtype of type 'num' of 'value'.
  numbers.setByKey('one', '1');
}

```

By indicating this generic as a parameter, we can avoid duplicating code and can continue getting the benefits of type annotations. You will see many examples of using generics in the Dart SDK. For example, the `List`, `Map`, and `Iterable` classes are good examples of using generics.

These data-type annotations are very useful and you can benefit from them by debugging your applications during design and programming phases, before they are executed. Data typing is truly something great and powerful! Now let's look at another useful tool, `typedef`.

Typedef

In order to provide a bit more information about functions in your application, you can use a `typedef`. A `typedef` essentially provides an alias for a function signature. Functions in Dart are first-class objects, which is why they may be passed as parameter to a function or be assigned to a variable. Let's see an example to better understand `typedefs`; we'll use a **callback** function that many JavaScript developers are accustomed to using.

A **callback** is a function passed as a parameter to another function. In the JavaScript world, due to memory leaks, the function is assigned to a variable, which is passed as a parameter. Here is a simple JavaScript example:

```

var out = console.log;
function login(user, password, out) {
  if (user == '' or password == '') {
    out('Access error. Enter username and password');
  }
}

```

Let's see a similar case in Dart and see what happens if we do not use `typedef`:

```

String formatLog(String msg, {int pos: 2}) {
  String cad = '';
  for(var i=0; i<pos; i++) {
    cad += '-';
  }
  return '${cad}${msg}';
}

```

```
void main() {
  var log = formatLog;
  var cad = 2 + log('My test message');
}
```

If you take a look at the example, you will notice that it has a clear error. In the line we've marked as bold, Dart Editor should alert you that you cannot concatenate a num with a String (the value returned by the formatLog function). In fact, if you run this example you'll get this exception on Dart Editor: type 'String' is not a subtype of type 'num' of 'other'.

Why didn't Dart Editor warn us about this while programming? Because we've assigned a function to a variable without data-type annotation, Dart Editor loses the data type of the original function. So, how can we avoid this?

The answer is simple: using **typedef**.

Typedef allows us to indicate that a function, although it's assigned to a variable or passed as a function parameter, should return a String type result.

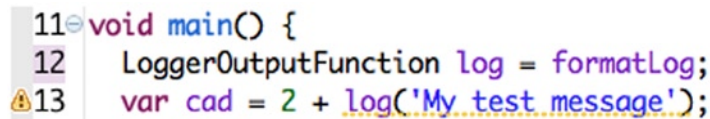
Let's see now how to use typedef to take advantage of type annotations, even if we assign the function to a variable:

```
String formatLog(String msg, {int pos: 2}) {
  String cad = '';
  for(var i=0; i<pos; i++) {
    cad += '-';
  }
  return '${cad}${msg}';
}

typedef String LoggerOutputFunction(String msg, {int pos});

void main() {
  LoggerOutputFunction log = formatLog;
  var cad = 2 + log('My test message');
}
```

Thanks to typedef we can get alerts from Dart Editor while we are designing and programming our application. Dart Editor will alert us that we cannot concatenate a String with a num.



```
11 void main() {
12   LoggerOutputFunction log = formatLog;
13   var cad = 2 + log('My test message');
```

Figure 20-1. Dart Editor displays the type warning thanks to the new typedef

Let's see how we can use typedefs to take advantage of data-type checking when we're passing a function as a parameter to another function. See this example:

```
void log(String msg) => print(msg);

num sum(num a, num b, logger) {
  logger(a);
  logger('+');
  logger(b);
  var res = a + b;
  logger('=');
  logger(res);
  return res;
}

void main() {
  sum(4, 9, log);
}
```

We now have two functions: `log` that prints out a message, and `sum` that adds two numbers. We're passing `log` as a parameter to `sum`. What's wrong with this code? If you run the example, you'll get an exception from Dart telling you that type `int` is not a subtype of type `String` of `msg`.

At the moment we passed the `log` function as a parameter to `sum`, Dart lost information about return types and parameter types for our `log` function. For this reason, we're not getting warnings while we're coding our application.

Let's see what happen if we add a typedef:

```
void log(String msg) => print(msg);

typedef void Logger(String msg);

num sum(num a, num b, Logger logger) {
  logger(a);
  logger('+');
  logger(b);
  var res = a + b;
  logger('=');
  logger(res);
  return res;
}

void main() {
  sum(4, 9, log);
}
```

In our new code, we've defined a typedef for the `log` function and used it in the `sum` function definition. With that simple code, Dart Editor can now warn us about the errors inside the `sum` function, as shown in Figure 20-2.


```

1 void log(String msg) => print(msg);
2
3 typedef void Logger(String msg);
4
5 num sum(num a, num b, Logger logger) {
6   logger(a);
7   logger('+');
8   logger(b);
9   var res = a + b;
10  logger('=');
11  logger(res);
12  return res;
13 }
14
15 void main() {
16   sum(4, 9, log);
17 }

```

Figure 20-2. Dart Editor warns us about issues inside the `sum` function thanks to `typedef`

Summary

In this chapter we have learned:

- What generics and typedefs are
- How we can use generics to reuse code and to take advantage of data-type checking
- How our classes can manage different data types using generics
- How to define typedef to avoid losing data-type annotations on functions when they're assigned to variables or passed as parameters to another functions



Using Dart's Libraries

In this chapter we'll cover Dart's libraries, what libraries are, and how to create your own libraries. We'll show you how to package functions, classes, or constants into a library for later use. Then we'll see how to create more complex libraries, conformed by multiple source files, and how to re-export a library so that it shows or hides some namespaces.

At the end of the chapter you'll see an example of how to use several concepts we've seen throughout this book, such as abstract classes, libraries, and pub packages. We have already seen how to define variables and functions, use the flow control and iteration statements, and create classes in Dart. This is much of what Dart offers.

Dart is a language that wants to become a standard for the development of next-generation web applications and therefore must present a mechanism to organize code, especially for large projects. In Dart, most of the time when you write an application, you will have multiple classes, some of which will inherit from others, and you will have different sections of your entire application in separate files, and so forth. Dart offers you the concept of a **library** to organize it all.

Now, in JavaScript we have a serious limitation, in that we have to work very hard to organize our entire project in code files. Libraries allow you to create modular code that can be easily shared and reused. A library is simply a package that you create from interfaces, classes, and global functions for reuse in another project using the keyword **library** followed by a name.

This is a simple example of creating a library, in which we have omitted the classes' implementation, or global functions bodies:

```
library animals;

num MAX_NUMBER = 12;

void createAnimal() {}

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}
```

■ **Note** The style guide recommends prefix library names with the package name and a dot-separated path.

So, according to this style guide, our previous library example would be called `chapter21.animals`, assuming the package's name is `chapter21`. See the example:

```
library chapter21.animals;

num MAX_NUMBER = 12;

void createAnimal() {}

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}
```

As you can see, you can pack everything you need into your library. You can add classes, global functions, or constants. A library can contain as many files as you need for solid organization of your code. In our example, our library will be formed only for one file, which is called `animals.dart`.

When a library has several files, they always have a file with the same name as the library. Let's see a more complex library example.

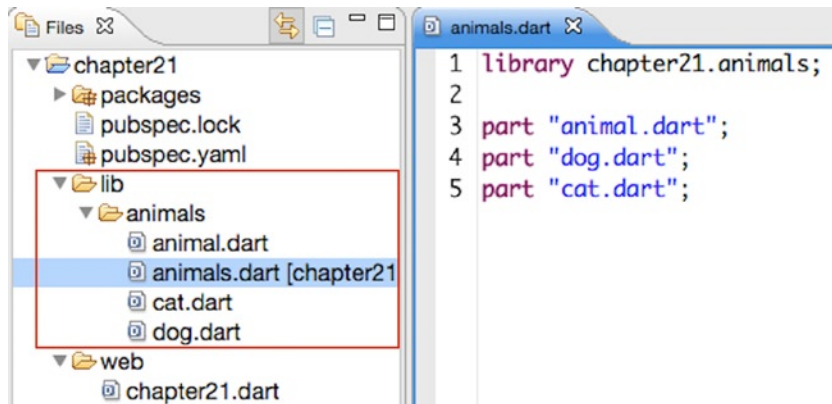


Figure 21-1. Creating a more complex library with several files

We'll create a new directory called `animals` in which to store our new library with all of its files. We'll place the new `animals` library inside `lib` directory according to Dart style guide. Then we'll create a main file with the same name as our library, `animals.dart`, which is the code of `animals.dart`.

```
library chapter21.animals;

part "animal.dart";
part "dog.dart";
part "cat.dart";
```

We'll indicate the name of the library prefixed with the package name, along with all the files that belong to this library. Then we'll create each file with its content. The `animal.dart` file will contain the `Animal` class.

```
part of chapter21.animals;

class Animal {

}
```

The `dog.dart` file will contain the `Dog` class. As you can see, on this file the first line will indicate that the file is part of the `animals` library by using the reserved keywords `part of`:

```
part of chapter21.animals;

class Dog extends Animal {

}
```

Finally the `cat.dart` file will contain the `Cat` class. As in our previous example, we avoid the implementations:

```
part of chapter21.animals;

class Cat extends Animal {

}
```

Using Libraries

Now that you are familiar with what libraries are and how to create them, let's see how you can work with them. Once you have defined a library that includes a series of interfaces, classes, functions, constants, and so forth, you can use it anywhere in your application simply by importing the library with the `import` keyword followed by the path of where it is stored.

We saw an example when talking about the library `dart:math` and learned to work with random numbers:

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

The only thing remarkable is that Dart-native libraries are imported using the name **dart:** followed by the library name. The same thing happens when you want to work with a package downloaded using `pub`:

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

You also can use this pattern when working with your own package or application. Any entry point (anything in `bin/`, `web/`, or another folder like `test`, `benchmark`, or `examples`) should also be able to import your own packages with the package uri. For example:

```
import 'package:chapter21/animals/animals.dart';
```

This should be used with the package-layout recommendation of putting libraries in the `lib/` directory. You will use the special name **package:** followed by the name or path of the library you want to use.

Regarding our previous `animals` library example, we could use this library using the `import` keyword followed by the path and the first file of the library, as follows:

```
import '../lib/animals/animals.dart';
```

```
void main() {
  // Using our library
  Dog d = new Dog();
  Cat c = new Cat();
  Animal a = new Animal();
}
```

As we've followed the package-layout recommendations, we also can import the library in this way:

```
import 'package:chapter21/animals/animals.dart';
```

```
void main() {
  // Using our library
  Dog d = new Dog();
  Cat c = new Cat();
  Animal a = new Animal();
}
```

When you work on your system, the most common problem you will run into is that you use many libraries, some of which may conflict with others because they have similar names. There are two different types of conflicts. The first involves internal library names (for example, you can't import two libraries each defined as `'library animal'`). The second conflict involves the symbols libraries introduce to the namespace (for example, there are two conflicting variables named `PI`, or two different classes named `Random`).

To fix these conflicts, you can specify a prefix for the library when you import it. Imagine you are working in a scientific application where you will use `dart:math` and another library, also called `math`. You could resolve the name conflict in this way:

```
import 'dart:math' as math;
import 'math.dart' as math_lib;
```

Later you can refer to any of the functions, classes, or constants of each library using the prefix you have defined at the time of importation.

```
math.PI;
math_lib.ALPHA;
```

At other times you may want to import only a part of a library. To do this you must use the keyword **show**, which indicates to Dart the part of that library to show. See the following:

```
import 'dart:convert' show JSON;
```

```
void main() {
  print(JSON.encode({'msg': 'Hello'})); // String: {"msg":"Hello"}
  print(JSON.decode({'msg':"Hello"})); // Map: {msg: Hello}
}
```

Dart also supports `hide` in order to import everything except a particular namespace. Let's see an example:

```
import 'dart:convert' hide JSON;
void main() {
  print(ASCII.encode("Hello")); // [72, 101, 108, 108, 111]
  // This will throw an exception because JSON is not defined.
  print(JSON.encode({'msg': 'Hello'}));
}
```

Creating Libraries with Multiple Files

At the beginning of this section we saw how to easily create a library. We created a single file called `animals.dart` with the `library` directive followed by its library name. Within this file we defined all the classes that make up our library. We've also seen that in Dart you can define libraries with multiple files. In the main library file you must specify which files compose the library with the `part` directive followed by the file name. In each file of your library you must also indicate the `part of` directive and its name in order to indicate that the file is part of that library.

Let's look at a simple example that follows the definition from our library of animals. Create a basic file that will be the `animals.dart` library with the indicated content, as follows:

```
library chapter21.animals;
```

```
part 'dogs.dart';
part 'birds.dart';
```

```
class Animal {}
```

Now create the files `dogs.dart` and `birds.dart`, making sure they include classes that define our dogs and our birds and are part of our `Animals` library.

The `dogs.dart` file looks like this:

```
part of chapter21.animals;
```

```
class Yorkshire extends Animal {}
class Beagle extends Animal {}
```

The `birds.dart` file will contain this code:

```
part of chapter21.animals;
```

```
class Canary extends Animal {}
class Blackbird extends Animal {}
```

Note that this example differs a little bit from the example we showed you at the beginning of this chapter. In this new example, we've created the `animals.dart` file with the `part` directives and added a class inside. No matter that this file is the main library file, as you can use it to add classes, global functions, or constants.

In addition, in this new example we've added several classes to each file belonging to the library. You can use these files to add all the code you need to your application. Now, from any part of your system or another application, you can use this library via the `import` statement followed by the path and name of the main library file. See the following:

```
import 'package:chapter21/animals.dart';

void main() {
  var ani = new Animal();
  var dog = new Yorkshire();
  var bird = new Canary();
}
```

Re-exporting Libraries

It may happen that you want to make a new library with similar functions as another library and you want to use it in another application. Or you want to create a small library from a subset of another library's functions. To do these types of things, Dart incorporates the export mechanisms that we will see below in a simple example.

From our `animals` library we're going to create a new library using the most interesting entries; we will package it and create a new library. Let's create a new file called `canine_mammals.dart` with this content:

```
library chapter21.canine_mammals.dart;

import 'package:chapter21/animals.dart';
export 'package:chapter21/animals.dart' show Animal, Yorkshire, Beagle;

class CanineMammal extends Animal {}
```

We have created a new library based on our previous `animals` library and re-exported the `Animal` base class and the two classes of dogs that we need. Note that the **show** clause is optional and that an entire library can be re-exported if desired by omitting this keyword.

Now we can use this new library in any application in this simple way:

```
import 'package:chapter21/canine_mammals.dart';

void main() {
  var ani = new Animal();
  var dog_y = new Yorkshire();
  var dog_b = new Beagle();
}
```

In this particular application, we cannot define objects of type `Canary` or `Blackbird` because they are not visible in our `canine_mammals` library.

EXERCISE: CREATING OUR FIRST LIBRARY USING MYSQL DRIVER

Throughout the following chapters we're going to show you different Dart code samples to reinforce your knowledge. At the end of the book, we'll put all it together to create a complete functional application. This application will let us manage our contacts.

To build off its functionalities, we'll use IndexedDb for local storage on the browser and MySQL for storing our contacts on MySQL server database, and we'll also use Future Asynchronous API, libraries, HTML5, CSS3, web services, and so on.

Now we're going to create our first library for this final project application. This library lets us work with a MySQL database to store our contacts. In this example, we're going to show you how to use abstract classes as well as how you can use external packages from Pub in your applications to best utilize Dart.

We're going to create a new command-line project called `chapter21_contacts`. After that we'll create the `contacts.dart` file inside the `lib/` directory. This new file will contain our library with `chapter21_contacts.mysql_contacts`.

In this library we'll have an abstract class and a concrete class for managing our contacts on MySQL. This is how our abstract class will look:

```
library chapter21_contacts.mysql_contacts;

import 'dart:async';
import 'package:sqljockey/sqljockey.dart';

/// Abstract class to manage contacts.
abstract class Contacts {
  /// Creates a new contact.
  Future add(Map data);

  /// Updates an existing contact.
  Future update(var id, Map data);

  /// Deletes an existing contact.
  Future delete(var id);

  /// Gets data for a given contact id.
  Future get(var id);

  /// Lists the existing contacts.
  Future list();

  /// Search for contacts,
  Future search(String query);
}
```


This abstract class defines the main methods by which to work with our contacts. We can add new contacts by passing a Map as a parameter with the new contact information to the add method.

```
{
  'fname': 'Moises',
  'lname': 'Belchin',
  'address': 'paseo del prado, 28',
  'zip': '28014',
  'city': 'Madrid',
  'country': 'Spain'
}
```

In our abstract class we also have an update method that is used to update the contact information; in this case, we'll pass as parameters the primary ID for the contact and a Map with the information to update.

The delete method lets us remove a contact from our database. In this case only the primary ID is necessary. We could retrieve a contact using a get method indicating the primary ID contact to retrieve.

Finally, we have developed two additional methods by which to get all the contacts from the database and search for contacts. Those methods are list and search.

As you can see, our abstract class Contacts will use Future API and the sqljocky package to work with MySQL.

■ **Note** You must install MySQL on your computer as well as sqljocky package from Pub. You can find the right version of MySQL for your system on <http://dev.mysql.com> Go to Chapter 6. Using Pub, the Dart Package Manager, learn how to install the sqljocky package. See here: <https://pub.dartlang.org/packages/sqljocky>

In the same contacts.dart file, we're going to develop our MySQLContacts class. This will be the concrete class we'll use to manage our contacts. It will look like this:

```
/// Concrete class to manage contacts stored on physical storage using MySQL.
/// This class requires sqljocky package.
/// https://pub.dartlang.org/packages/sqljocky
///
/// MySQL DB structure:
///
/// CREATE DATABASE `dbContacts` ;
/// CREATE TABLE `dbContacts`.`contacts` (
///   `id` INT( 11 ) NOT NULL AUTO_INCREMENT ,
///   `fname` VARCHAR( 150 ) NOT NULL ,
///   `lname` VARCHAR( 150 ) NULL ,
///   `address` TEXT NULL ,
///   `zip` VARCHAR( 10 ) NULL ,
///   `city` VARCHAR( 150 ) NULL ,
///   `country` VARCHAR( 150 ) NULL ,
///   PRIMARY KEY ( `id` )
/// ) ENGINE = MYISAM ;
class MySQLContacts extends Contacts {
```

```

/// MySQL sqljocky database object.
ConnectionPool _db;
/// MySQL host.
String _host;
/// MySQL port.
int _port;
/// Database name.
String _dbName;
/// MySQL default table name.
String _tableName;
/// MySQL User name.
String _user;
/// MySQL Password.
String _password;

/// Constructor.
MySQLContacts({String host:'localhost', int port:3306,
               String user:'root', String password:'',
               String db:'dbContacts', String table:'contacts'}) {
  _host = host;
  _port = port;
  _dbName = db;
  _tableName = table;
  _user = user;
  _password = password;

  if(_password != null && _password.isNotEmpty) {
    _db = new ConnectionPool(host:_host, port:_port, user:_user,
                           password:_password, db:_dbName, max:5);
  } else {
    _db = new ConnectionPool(host:_host, port:_port, user:_user,
                           db:_dbName, max:5);
  }
}

/// Deletes all the information on the `contacts` table.
Future dropDB() {
  return _db.query("TRUNCATE TABLE $_tableName").then((_) => true);
}

/// Closes DB connection.
void close() => _db.close();

/// Creates a new contact.
Future add(Map data) {
  var c = new Completer();
  var fields = data.keys.join(",");
  var q = data.keys.map((_) => '?').join(", ");

```

```

    var values = data.values.toList();
    _db.prepare("INSERT INTO $_tableName ($fields) VALUES ($q);").then((query) {
      c.complete(query.execute(values));
    });
    return c.future;
  }

```

/// Updates an existing contact.

```

Future update(int id, Map data) {
  var c = new Completer();
  var fields = data.keys.map((v) => "$v = ?").join(", ");
  var values = data.values.toList();
  values.add(id);
  _db.prepare("UPDATE $_tableName SET $fields WHERE id = ?;").then((query) {
    c.complete(query.execute(values));
  });
  return c.future;
}

```

/// Deletes an existing contact.

```

Future delete(int id) {
  var c = new Completer();
  _db.prepare("DELETE FROM $_tableName WHERE id = ?;").then((query) {
    c.complete(query.execute([id]));
  });
  return c.future;
}

```

/// Gets data for a given data or null if not exist.

```

Future get(int id) {
  var element;
  return _db.prepare("SELECT * FROM $_tableName WHERE id = ?")
    .then((query) => query.execute([id]))
    .then((result) => result.forEach((row) {
      element = {
        'fname': row.fname,
        'lname': row.lname,
        'address': row.address,
        'zip': row.zip,
        'city': row.city,
        'country': row.country;
      });
    })
    .then((_) => element);
}

```

/// Lists the existing contacts.

```

Future list() {
  var results = [];
  return _db.query("SELECT * FROM $_tableName")
    .then((rows) => rows.forEach((row) {
      results.add({

```

```

        'id': row.id,
        'fname': row.fname,
        'lname': row.lname,
        'address': row.address,
        'zip': row.zip,
        'city': row.city,
        'country': row.country
    });
  });
  .then((_) => results);
}

/// Search for contacts,
Future search(String query) {
  query = query.toLowerCase().trim();
  var matches = [];
  var where = "" "LOWER(fname) like '%$query%' OR
  LOWER(lname) like '%$query%' OR LOWER(address) like '%$query%' OR
  LOWER(zip) like '%$query%' OR LOWER(city) like '%$query%' OR
  LOWER(country) like '%$query%'"";

  return _db.query("SELECT * FROM $_tableName WHERE $where")
    .then((rows) => rows.forEach((row) {
      matches.add({
        'id': row.id,
        'fname': row.fname,
        'lname': row.lname,
        'address': row.address,
        'zip': row.zip,
        'city': row.city,
        'country': row.country
      });
    }));
  .then((_) => matches);
}
}

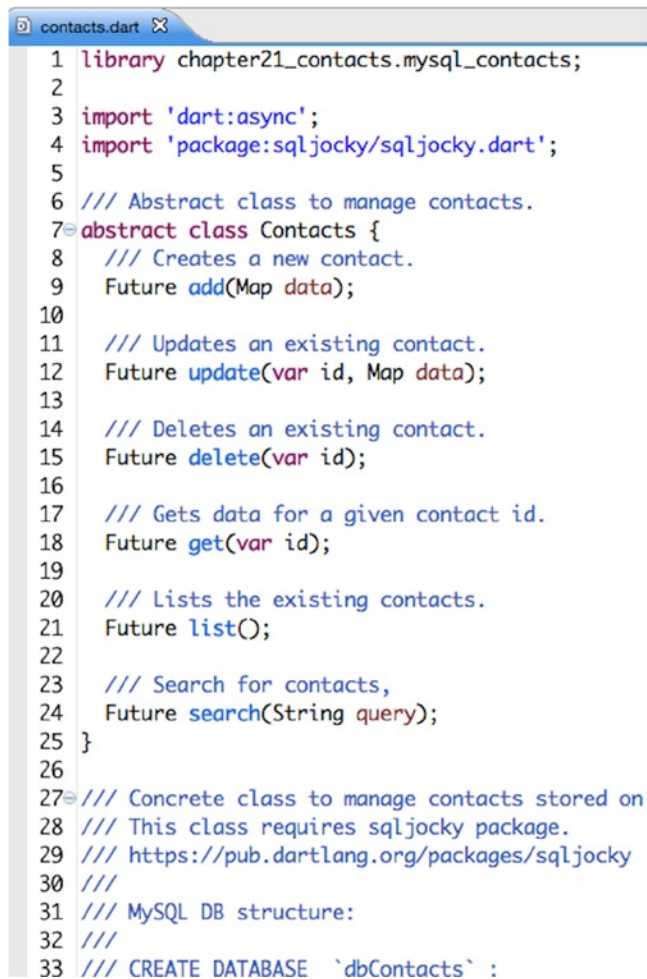
```

Our concrete `MySQLContacts` class extends from our abstract `Contacts` class and implements all its methods; in addition, this class adds the necessary instance variables and its constructor method.

All the methods of this class will return a `Future` object to make our code asynchronous, which means that we are not waiting for the MySQL operations to finish their jobs. We'll see the `Future` class and asynchronous programming more in depth in coming chapters.

We now have the abstract class and the concrete class, and to properly use them we're going to put them in a library called `sql_contacts`.

This class will import `dart:async` and `sqljockey` package:



```

1 library chapter21_contacts.mysql_contacts;
2
3 import 'dart:async';
4 import 'package:sqljockey/sqljockey.dart';
5
6 /// Abstract class to manage contacts.
7 abstract class Contacts {
8   /// Creates a new contact.
9   Future add(Map data);
10
11   /// Updates an existing contact.
12   Future update(var id, Map data);
13
14   /// Deletes an existing contact.
15   Future delete(var id);
16
17   /// Gets data for a given contact id.
18   Future get(var id);
19
20   /// Lists the existing contacts.
21   Future list();
22
23   /// Search for contacts,
24   Future search(String query);
25 }
26
27 /// Concrete class to manage contacts stored on
28 /// This class requires sqljockey package.
29 /// https://pub.dartlang.org/packages/sqljockey
30 ///
31 /// MySQL DB structure:
32 ///
33 /// CREATE DATABASE `dbContacts` ;

```

Figure 21-2. Creating our `chapter21_contacts.mysql_contacts` library with `Contacts` and `MySQLContacts` classes

We'll see now how to use this class and store our contacts on MySQL database server with Dart and `sqljockey`.

■ **Note** Remember you must have MySQL server running on your system before running our Dart application, and you also need to create the database structure.

You could create the required database structure by running this sequence on MySQL server:

```
CREATE DATABASE `dbContacts` ;
CREATE TABLE `dbContacts`.`contacts` (
  `id` INT( 11 ) NOT NULL AUTO_INCREMENT ,
  `fname` VARCHAR( 150 ) NOT NULL ,
  `lname` VARCHAR( 150 ) NULL ,
  `address` TEXT NULL ,
  `zip` VARCHAR( 10 ) NULL ,
  `city` VARCHAR( 150 ) NULL ,
  `country` VARCHAR( 150 ) NULL ,
  PRIMARY KEY ( `id` )
) ENGINE = MYISAM ;
```

The following is an example to show you how to create a new contact, update its information, and delete it.

```
import 'package:chapter21_contacts/contacts.dart' show MySQLContacts;
void main() {

  // Managing contacts
  var id;
  var myc = new MySQLContacts();

  // Cleanup database and adding new contact.
  myc.dropDB().then((_) => myc.add({'fname': 'Moises'}))
    .then((contact) {
      id = contact.insertId;
      print(id);
      // getting info for the give ID contact
      return myc.get(id);
    })
    .then((contact) {
      print(contact);
      // updating contact info
      return myc.update(id, {'fname': 'Moisés', 'lname': 'Belchín'});
    })
    .then((_) {
      print('updated');
      // showing changes after updating.
      return myc.get(id);
    })
    .then((contact) {
      print(contact);
      // Deleting the contact.
      return myc.delete(id);
    })
    .then((_) {
      // getting a nonexistent key will return NULL
      return myc.get(id);
    })
}
```

```

    .then((contact) {
      print(contact); // NULL
      // Close DB connection.
      myc.close();
    });
  }
}

```

After you run this code, you should see result on the Output view window similar to the following:

```

1
{fname: Moises, lname: null, address: null, zip: null, city: null, country: null}
updated
{fname: Moisés, lname: Belchín, address: null, zip: null, city: null, country: null}
null

```

Now we'll show you how to add multiple contacts to your databases and list all the information:

```

import 'dart:async';
import 'package:chapter21_contacts/contacts.dart' show MySQLContacts;

void main() {

  var myc = new MySQLContacts();

  // Adding multiple contacts
  Future.wait([
    myc.dropDB(),
    myc.add({
      'fname': 'Moises',
      'lname': 'Belchin',
      'address': 'paseo del prado, 28',
      'zip': '28014',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    myc.add({
      'fname': 'Patricia',
      'lname': 'Juberias',
      'address': 'Castellana, 145',
      'zip': '28046',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    myc.add({
      'fname': 'Peter',
      'lname': 'Smith',
      'address': 'Cyphress avenue',
      'zip': '11217',
      'city': 'Brooklyn',
      'country': 'EEUU'
    }),
  ]).then((_) {

```

```

// Listing all the contacts from our database.
myc.list().then((results) {
  results.forEach((r) => print(r));
  myc.close();
});

});
}

```

After running this code you should see something similar to the following:

```

{id: 1, fname: Moises, lname: Belchin, address: paseo del prado, 28, zip: 28014, city:
Madrid, country: Spain}
{id: 2, fname: Peter, lname: Smith, address: Cyphress avenue, zip: 11217, city: Brooklyn,
country: EEUU}
{id: 3, fname: Patricia, lname: Juberias, address: Castellana, 145, zip: 28046, city: Madrid,
country: Spain}

```

Finally, you could use this class to search for some contacts on your MySQL database server:

```

import 'dart:async';
import 'package:chapter21_contacts/contacts.dart' show MySQLContacts;
void main() {
  var myc = new MySQLContacts();
  Future.wait([
    myc.dropDB(),
    myc.add({
      'fname': 'Moises',
      'lname': 'Belchin',
      'address': 'paseo del prado, 28',
      'zip': '28014',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    myc.add({
      'fname': 'Patricia',
      'lname': 'Juberias',
      'address': 'Castellana, 145',
      'zip': '28046',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    myc.add({
      'fname': 'Peter',
      'lname': 'Smith',
      'address': 'Cyphress avenue',
      'zip': '11217',
      'city': 'Brooklyn',
      'country': 'EEUU'
    }),
  ]).then((_) {

```



```

// Searching for contacts located in Madrid
var query1 = 'madrid';
myc.search(query1).then((results) {
  print('Looking for: $query1');
  if(results.length <= 0) {
    print('No results found');
  } else {
    results.forEach((r) => print(r));
  }
  myc.close();
});

// Searching for contacts which contains `cyp`
var query2 = 'cyp';
myc.search(query2).then((results) {
  print('Looking for: $query2');
  if(results.length <= 0) {
    print('No results found');
  } else {
    results.forEach((r) => print(r));
  }
  myc.close();
});
});
}

```

After executing this code, you could see something similar to these results:

```

Looking for: madrid
{id: 2, fname: Moises, lname: Belchin, address: paseo del prado, 28, zip: 28014, city:
Madrid, country: Spain}
{id: 3, fname: Patricia, lname: Juberias, address: Castellana, 145, zip: 28046, city: Madrid,
country: Spain}

```

```

Looking for: cyp
{id: 1, fname: Peter, lname: Smith, address: Cyphress avenue, zip: 11217, city: Brooklyn,
country: EEUU}

```

Summary

In this chapter we have learned:

- What Dart's libraries are
- How to create your own libraries in Dart
- How to create more complex libraries with multiple files
- How to use your own libraries on your projects
- How to re-export libraries showing or hiding some namespaces of the original library



Leveraging Isolates for Concurrency and Multi-Processing

In this chapter we'll cover what isolates are and how you can use them—along with concurrency and multi-processing—to run your Dart programs. We'll learn everything about isolates through two simple examples so that you will know how to use multi-processing in your programs. Dart also lets you work concurrently and use all multi-processing capabilities, thanks to the multi-core in our modern computers.

In other programming languages this is known as *threads*. The difference between the threads in other languages and the Dart isolates is that the latter do not share memory; each isolate is independent of other isolates, and they communicate through messages.

When threads share memory, the application is prone to errors and the code becomes very complicated, making debugging difficult. Each Dart's isolate has its own memory and executes in an isolated way from the rest, hence their name. Isolates are possible in Dart and in the JavaScript translation because web workers are available in HTML5. Web workers are JavaScript tasks that run in the background to avoid the webpage being locked until a process finishes. When Dart code is compiled to JavaScript, `dart2js` use web workers to operate, although the ideal is to use *futures* and *streams* (more on those later).

Here you can see a simple example of Dart isolates:

```
import 'dart:async';
import 'dart:isolate';

void CostlyProcess(SendPort replyTo) {
  var port = new ReceivePort();
  replyTo.send(port.sendPort);
  port.listen((msg) {
    var data = msg[0];
    replyTo = msg[1];
    if (data == "START") {
      replyTo.send("Running costly process....");
      // Costly stuff here
      replyTo.send("END");
      port.close();
    }
  });
}
```

```

void main() {
    print('Start application');

    // Create reply port for the isolate.
    var reply = new ReceivePort();

    // We'll run the costly process in an Isolate.
    Future<Isolate> iso = Isolate.spawn(CostlyProcess, reply.sendPort);
    iso.then((_) => reply.first).then((port) {

        // Once we've created the Isolate, we send the start message process and
        // we're waiting to receive a reply from the costly process.
        reply = new ReceivePort();
        port.send(["START", reply.sendPort]);
        reply.listen((msg) {
            print('Message received from isolate: $msg');
            if (msg == "END") {
                print('Costly process completed successfully !!!');
                reply.close();
            }
        });

    });

    // We can continue executing more instructions while we're waiting for the
    // execution reply of our costly process in background.
    print ('Continue running instructions');
}

```

We've defined a function called `CostlyProcess`, which represents a costly process that will be executed by taking advantage of the concurrent programming. As you can see, this function uses ports—`SendPort` and `ReceivePort`—to communicate with the main process.

`SendPorts` are created from `ReceivePorts`. Any message sent through a `SendPort` is delivered to its corresponding `ReceivePort`. There could be many `SendPorts` for the same `ReceivePort`. `SendPorts` can be transmitted to other isolates, and they preserve equality when sent.

The `send()` method for the `SendPort` class sends an asynchronous message through this `SendPort` to its corresponding `ReceivePort`. The content of the message can be primitive values (`null`, `num`, `bool`, `double`, `String`), instances of `SendPort`, and lists or maps whose elements are any of these.

Later, we created the `ReceivePort` to communicate with our costly process resulting from the main function, executing the costly function in a separate isolate using the `Isolate.spawn` function.

`ReceivePorts` have a `SendPort` getter, which returns a `SendPort`. Any message that is sent through this `SendPort` is delivered to the `ReceivePort` it was created from. There, the message is dispatched to the `ReceivePort`'s listener.

A `ReceivePort` is a non-broadcast stream. This means that it buffers incoming messages until a listener is registered. Only one listener can receive messages. If you run the above process you will get this result:

```

Start application
Continue running instructions
Message received from isolate: Running costly process....
Message received from isolate: END
Costly process completed successfully !!!

```

In this case, we’ve shown how to run a process in concurrent processing. The process we’re executing is available in the same file where we have the **main()** function. Dart also allows you to run concurrent processes using isolates located in different source files.

Let’s look at a simple example of a file structure like this. The `chapter22.html` file has this content. It is the basic example that Dart Editor loads when you select “Build a sample application code.” See the following:

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Chapter22 isolates</title>

    <script async type="application/dart" src="chapter22_isolates.dart"></script>
    <script async src="packages/browser/dart.js"></script>

    <link rel="stylesheet" href="chapter22.css">
  </head>
  <body>
    <h1>Chapter22 isolates</h1>

    <p>Hello world from Dart!</p>

    <div id="sample_container_id">
      <p id="sample_text_id"></p>
    </div>

  </body>
</html>
```

We also have the `chapter22.css` file with this CSS style:

```
body {
  background-color: #F8F8F8;
  font-family: 'Open Sans', sans-serif;
  font-size: 14px;
  font-weight: normal;
  line-height: 1.2em;
  margin: 15px;
}

h1, p {
  color: #333;
}

#sample_container_id {
  width: 100%;
  height: 400px;
  position: relative;
  border: 1px solid #ccc;
  background-color: #fff;
}
```

```
#sample_text_id {
  font-size: 15pt;
  text-align: center;
  margin-top: 140px;
  -webkit-user-select: none;
  user-select: none;
}
```

Then we have a `chapter22_isolates.dart` file with this code:

```
import 'dart:isolate';
import 'dart:html';
import 'dart:async';

main() {
  Element myresult = querySelector('#sample_text_id');

  // Define send/reply ports.
  SendPort sendPort;
  ReceivePort receivePort = new ReceivePort();
  receivePort.listen((msg) {
    if (sendPort == null) {
      sendPort = msg;
    } else {

      switch(msg) {
        case "FROM PROCESS: START":
          myresult.appendHtml(' Running costly process....<br/><br/>');
          break;

        case "FROM PROCESS: END":
          myresult.appendHtml(' Costly process completed successfully<br/><br/>');
          receivePort.close();
          break;

        default:
          myresult.appendHtml(' Received from the isolate: $msg<br/><br/>');
      }
    }
  });

  // This is a costly process that we want to execute.
  String Costlyprocess = 'costly_process.dart';
  int counter = 0;
  var timer;
  Isolate.spawnUri(Uri.parse(Costlyprocess), [], receivePort.sendPort).then((isolate) {
    print('Isolate running !!');
    timer = new Timer.periodic(const Duration(seconds: 1), (t) {
```

```

switch (counter) {
  case 0:
    sendPort.send("START");
    break;

  case 5:
    sendPort.send("END");
    timer.cancel();
    break;

  default:
    // Every second we send a message to the costly process.
    sendPort.send('From application: ${counter}');
}
counter += 1;

});
});

// We can continue executing more code while waiting for the
// reply execution of our process in background.
print('Continue executing more instructions');
print('.....');
print('Application completed');
}

```

Finally, we have another file called **costly_process.dart** that contains our costly process code, and this is what we'll call from the new isolate created on **chapter22_isolate.dart**:

```

import 'dart:isolate';

main(List<String> args, SendPort sendPort) {
  ReceivePort receivePort = new ReceivePort();
  sendPort.send(receivePort.sendPort);
  receivePort.listen((msg) {
    sendPort.send('FROM PROCESS: $msg');
  });
}

```

In this case we have the `main()` function in one file and the costly process in another one. In the main file we've used a timer to send information each second from `main()` to `costly_process`. If you run this code in the Dart Editor Output view you will see the text of the print statements we have made in the code:

```

Continue executing more instructions
.....
Application completed
Isolate running !!

```

Figure 22-1 is the window in which you will see this result executed second by second.

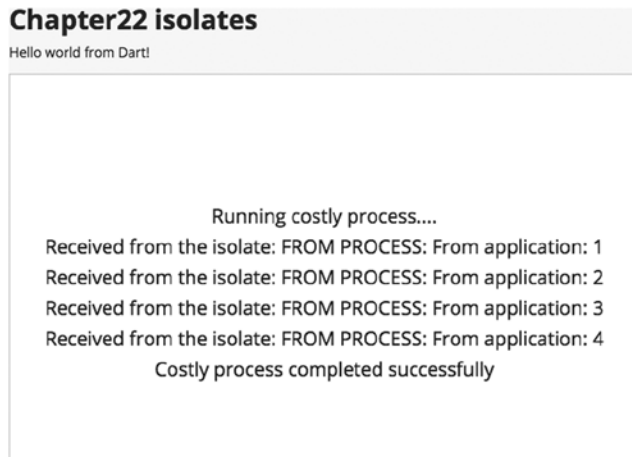


Figure 22-1. The result of the communication between *chapter22_isolates.dart* file and the *costly_process.dart* file

The examples you’ve seen above don’t do anything amazing, but they’re perfect for teaching you how to configure your environment and play with concurrent programming in Dart.

Summary

In this chapter we have learned:

- What isolates are and how they allow you to use concurrent programming in Dart
- How to make an isolate run in concurrent-mode costly processes
- What `ReceivePort` and `SendPort` are
- How to use multi-threading in Dart to run costly processes located in either the same code file or different files



Asynchronous Programming with Dart

In this chapter we'll cover the asynchronous programming in Dart, what Future and Stream are and how you can work with them. We'll see how you can create your own asynchronous functions in Dart using the Future API and Completers.

At the end of the chapter we'll show you an example of creating your own library based on Future API to manage contact information stored in IndexedDb.

When we began working with Dart we noticed this language would have great potential, not only for the great tools it offers but also how this new language unify all the mixture of tools, languages , and browsers we use to make web development.

Also it currently includes APIs that we can use and they're not available in JavaScript until the new language version is published.

If we consider this upcoming version 6 of JavaScript, which we hope will be available by the end of 2014, there are many functions and APIs already available in Dart, including:

- Block scope variables.
- Default values in functions.
- Optional Parameters by name.
- String Interpolation.
- Arrow functions.
- Promises API.

And these are just a few examples.

What is Asynchronous Programming?

Asynchronous programming, as presented by Dart and JavaScript 6 with its API Promises, is a way to work, creating and executing asynchronous processes in a natural, simple, and structured way. Dart offers the `dart:async` API with Future and Stream classes that you can use to do asynchronous programming. Actually, is essential that you understand what Future and Stream are and how they work because this is a common way to work in Dart's SDK. When you begin running and using them you will realize that these classes have a huge potential. They offer a very structured way to work asynchronously and avoid the callbacks hell very common in JavaScript.

Future

A Future object represents a process, function, or return value to be recovered in a future period of time. We will execute a function, for example, and we don't get immediately the return value of the function, we will get it in another moment during the application execution. Consider how a website works with links to other documents. The browser renders the document and displays it to the user. The user can start reading a text within two minutes and then click on a link. If you want to know when the user clicks on that link, you must capture the click event of the user. The event is the result of listening on that link (Future) until the user clicks it. When the user clicks on the link, you get a result at a future time that could be 5 seconds or 45 minutes from the web page loading, depending on what the user is doing in the website.

Except for the few minor differences, that is how a Future object works in Dart. You execute a process that does some action and you await the results. Best of all, while waiting for the result, your application is not stuck, it's continuously running more actions and your users don't notice anything. Using the Future class you can handle these objects and add asynchronous functions to your applications.

Working with Future

Let's look at some examples of Future and how we can work with them. We have commented that in the Dart SDK you can find many classes returning Future objects as an execution result. In `dart:html` you can find an example of this. We're going to use the `HttpRequest` class located on `dart:html` to make asynchronous requests and work with the results of these requests.

```
import 'dart:html';

void main() {
  print('1. Run a request asynchronously ');
  var result = HttpRequest.getString('response.json');
  result.then((future_result) {
    print('2. The Future has been completed and I get the result.');
```

```
    print(future_result);
  }).catchError((_) {
    print('3 Error running the request.');
```

```
  });
  print('4. I continue running more actions.');
```

```
}
```

To run this example properly you have to create a file called `response.json` with this content.

```
{'message': 'Welcome to the asynchronous programming with Dart'}
```

If you run this little application you'll get these messages.

```
1. Run a request asynchronously
4. I continue running more actions.
2. The Future has been completed and I get the result.
{'message': 'Welcome to the asynchronous programming with Dart'}
```

As you can see, regardless of the message we display, our little application is not waiting for the response of the `HttpRequest.getString`, it continues running more actions and when the server responds with a valid value or with an error, then the `.then` or `.catchError` methods of the returned `Future` object are executed. In the `Future` class you'll find interesting methods to work with these objects and it is imperative that you learn them to master the asynchronous programming.

First, we have several constructors to create `Future` objects. Using the default **Future** constructor we can create a future object containing the result of calling a computation asynchronously with `Timer.run`. Let's see an example of this constructor.

```
import 'dart:async';

int myfunc() {
  print('Making a lot of stuff here');
  print('...');
  print('...');
  var result = 159;
  return result;
}

void main() {
  print('1');
  var fut = new Future(myfunc);
  fut.then((r) => print('Result of myfunc: $r'));
  for(var i = 0; i<5; i++) {
    print('making other tasks');
  }
}
```

We've created `myfunc`, a function that will be the computation of the new `Future`. On the `main` method we've created a future object with `new Future myfunc` function as parameter. This creates a `Future` object and schedules a `Timer` to execute the function we give as an argument. If you run the above code you'll get this result.

```
1
making other tasks
making other tasks
making other tasks
making other tasks
making other tasks
Making a lot of stuff here
...
...
Result of myfunc: 159
```

We can see the first statement, later we've created the `Future` object and we continue executing other statements or tasks. As soon as the results of `new Future(myfunc)` are available we'll get it.

Into the Future class we have `Future.microtask` which is very similar to the Future constructor. It creates a future containing the result of calling a computation asynchronously with `scheduleMicrotask`. Let's see an example.

```
import 'dart:async';
void main() {
  print('1');
  var fut = new Future.microtask(() => 12);
  fut.then((v) => print('Result: $v'));
  print('2');
}
```

After running this example you can see this output.

```
1
2
Result: 12
```

As you can see the print statements are executed first, then when the future object is ready we'll see the `Result: 12` statement. Future API has another constructor `Future.value` this constructor creates a future object whose value is available in the next event-loop iteration. Let's see an example.

```
import 'dart:async';
void main() {
  print('1');
  var fut = new Future.value(12);
  fut.then((v) => print('Result: $v'));
  print('2');
}
```

If you run this code you can see these results:

```
1
2
Result: 12
```

You'll see at first the print statements and in the next event-loop iteration you'll see the result of `Future.value`.

We have another interesting constructor that allows us to create future objects that run its computation after a delay. Let's see an example.

```
import 'dart:async';

int costly_function() => 12;

void main() {
  print('start');
  var fut = new Future.delayed(new Duration(seconds:5), costly_function);
  fut.then((v) => print(v));
  for(var i=0; i<8; i++) {
    print('Other tasks');
  }
  print('end');
}
```

If you run that program you can see all the print statements and after 5 seconds you can see how Dart shows you in the Output Tools view the result of the `costly_function`.

```
start
Other tasks
Other tasks
Other tasks
Other tasks
Other tasks
Other tasks
Other tasks
Other tasks
end
12
```

You have the static method `wait` to launch multiple asynchronous functions and await the result of all of them.

```
var futures = [HttpRequest.getString('response.json'),
               HttpRequest.getString('response.json'),
               HttpRequest.getString('response.json')];
Future.wait(futures).then((response) {
  response.forEach((e) => print(e));
});
```

Note Remember if you want to execute these examples you must import `dart:html` and `dart:async` in your dart applications.

We've seen `.then` method in our first example. This method lets you get the result returned when the `Future` object is completed and returns a value.

```
var fut = HttpRequest.getString('http://www.google.es');
fut.then((response) => handleResponse(response));
```

Before we also saw the `.catchError` method, which is very useful to detect if an error occurs during the execution of the asynchronous function.

```
var fut = HttpRequest.getString('http://www.google.es');
fut.catchError((error) => handleError(error));
```

We've seen the `.catchError` method to handle errors produced with Futures but if you need to complete your own futures with an error you can use the new `Future.error` constructor. This method creates a future object that completes with an error in the next event-loop iteration. Let's see an example.

We're going to create a future function and that function will make some parameter validations. If we have a problem with parameters we'll throw an `Exception`. As we are developing a future function but we have to throw the exceptions with `Future.error` constructor, so the `.catchError` will properly handle the errors.

```
import 'dart:async';

Future costly_calculation(Map parameters) {
  if(parameters.isEmpty) {
    return new Future.error(new Exception('Missing parameters'));
  }
  print('Doing costly calculation...');
  print('At some point we have an error');
  return new Future.value(12);
}

void main() {
  costly_calculation({})
    .then((v) => print('Result: $v'))
    .catchError((error) => print('Error: $error'));
}
```

The `.whenComplete` method allows you to register a function that will execute when the object `Future` has completed its task, whether it is completed successfully or not.

```
var fut = HttpRequest.getString('http://www.google.es');
fut.whenComplete((()) => handleTask ());
```

The `.asStream` method allows you to generate a `Stream` from the `Future` object.

```
var fut = HttpRequest.getString('response.json');
fut.asStream().listen((v) => print(v));
```

Finally, you have the `.timeout` method to register a function to be executed after a specified period of time. For example, if you want to get a resource from the Internet and your connection is disrupted, the future won't wait forever; after 10 seconds it will print out the `TimeOUT!` message.

```
var fut = HttpRequest.getString('response.json');
fut
  .then((v) => print(v))
  .timeout(new Duration(seconds:10), onTimeout: () => print('TimeOUT!'));
```

Before finishing this section we want to talk about chaining futures. In some circumstances you need to execute some future functions and with the result of the first future function work in the second one and so on. You have two ways of doing that, as described in the following section.

```
import 'dart:async';

Future<int> one() => new Future.value(1);
Future<int> two() => new Future.value(2);
Future<int> three() => new Future.value(3);

void main() {
  one().then((v) {
    print('Result from one: $v');
    two().then((v) {
      print('Result from two: $v');
      three().then((v) {
        print('Result from three: $v');
        print('End');
      });
    });
  });
}
```

In this example we start executing `one()` when this future is completed then we execute `two()` when it's completed execute `three()`. Nested calls work but they're harder to read. We can chain calls instead.

```
import 'dart:async';

Future<int> one() => new Future.value(1);
Future<int> two() => new Future.value(2);
Future<int> three() => new Future.value(3);

void main() {
  one().then((oneValue) {
    print('Result from one: $oneValue');
    return two();
  }).then((twoValue) {
    print('Result from two: $twoValue');
    return three();
  }).then((threeValue) {
    print('Result from three: $threeValue');
    print('End');
  });
}
```

When Future-returning functions need to run in order, use chained `then()` calls because they're more readable.

Stream

A `Stream` object is similar to a `Future` object. The only difference is that it is designed for repetitive data sets, but conceptually it is like a `Future` object in that it will return the result of an asynchronous execution at a particular time. For this reason you can convert a `Stream` into a `Future` and a `Future` into a `Stream`. Actually `Stream` is the class that you use when you add an event handler on a button in an HTML document in `dart:html`. In fact all the HTML event management applications use this class. It is also used for managing files and directories on `dart:io`, open, create, copy files, or create directories and list their contents.

Working with Stream

Let's look at some examples of event management in `dart:html` and how to work with `Stream` when we register a new event handler.

We'll start from an HTML document where we'll add a button and create a handler for the button click event. We'll stay listening to this event type in our button using the `Stream` class.

Each time the user clicks on this button, we will get a `StreamSubscription` object that will allow us to obtain the event that occurs in the HTML element, cancel the listening to this event, or pause it temporarily.

Here are some examples of using `Stream` and `StreamSubscription` in `dart:html` over `ButtonElement`.

```
var button = document.querySelector('#button');

button.onClick.listen((event) {
  print('Button clicked !!!');
});
```

With the `.listen` method of the `Stream` class we can create a new subscription and every time the user clicks the button, the function `onData` of `StreamSubscription` will be executed displaying the message `Button clicked !!!`.

In addition to the `onData` function, this method provides the `onError`, `onDone`, and `cancelOnError` functions, which as you can imagine register functions to be executed when an error occurs in the `Stream`, when the `Stream` is closed the `onDone` method will be executed and finally `cancelOnError` allows you to finish the subscription when an error occurs.

If you take a look at the `Stream` class you will see a lot of methods that we saw in `dart:core` when we worked with `Iterable`. These methods on `Stream` class are similar to the `Iterable` methods designed for repetitive data sets. In other words, a `Stream` allows you to work with each element placed in the `Stream`, so there are methods such as `where`, `map`, `fold`, `join`, `forEach`, `every`, `any`, `last`, `first`, and `single`.

```
void main() {

  print('1. Actions');
  var serie = [1, 2, 3, 4, 5];
  var stream = new Stream.fromIterable(serie);

  // We subscribe to the events of Stream
  stream.listen((value) {
    print("2. Stream: $value");
  });
  print('3. More actions');
}
```

After executing this code you'll get this result.

```
1. Actions
3. More actions
2. Stream: 1
2. Stream: 2
2. Stream: 3
2. Stream: 4
2. Stream: 5
```

As you can see, the result is similar to the execution of the `Future` examples. Our main program is not waiting for the `Stream`; it is completely asynchronous and when the data is received from the `Stream` the `onData` registered function is executed.

Creating Asynchronous Functions

We've learned the `Future` and `Stream` classes and it's time to create and use your own asynchronous functions and take advantage of all their functionality. A class that we have not discussed so far and is available in `dart:async` is `Completer`, which lets you to create `Future` objects and complete them later with a particular value or failed, as appropriate. We're going to create a very simple function but we're going to make it fully asynchronous. With this simple example you will understand how to make asynchronous applications.

```
import 'dart:async';

Future getUser() {
  final c = new Completer();
  c.complete({'user': 'Moises'});
  return c.future;
}

void main() {
  print('1. Doing some stuff');
  var u = getUser();
  u.then((v) {
    print('2. Result future.');
    print(v['user']);
  });
  print('3. Doing more stuff');
}
```

First we create a function that returns a `Future` object. In this function we use the `Completer` class to create a `Completer` object which will return value and we return the future of the `Completer` object. Then, from our application we call that function and we use the `Future` objects methods to work with it.

If you run this code you can see how the function is executed asynchronously the main program continues executing more actions and then, when the future value is ready we receive it and print it out.

```
1. Doing some stuff
3. Doing more stuff
2. Result future.
Moises
```

While `completer`'s are extremely powerful for creating a `Future`-based API, they are often more complex than required for the majority of `Future`-based APIs. It is recommended to use `Future` constructors and chains. Let's see how we can make the `getUser()` function asynchronous without using `Completer` class.

```
Future getUser() => new Future.value({'user': 'Moises'});

void main() {
  print('1. Doing some stuff');
  var u = getUser();
  u.then((v) {
    print('2. Result future.');
    print(v['user']);
  });
  print('3. Doing more stuff');
}
```


In this case we can use one of the constructors we've seen in the previous section such as `new Future.value` to make our function asynchronous without using Completer. Completers are beneficial primarily when converting a callback based API to a Future based one (such as when creating a wrapper).

EXERCISE: CREATING AN ASYNCHRONOUS CLASS USING INDEXEDDB AND GOOGLE MAPS LIBRARIES

As we mentioned in *Chapter 21* you can find a big project to manage your contacts information. In that chapter we developed a class to store our contacts on the MySQL database server. We saw abstract classes and how to extend from them to create a more concrete class and how to use libraries to give more potential to our Dart applications.

We already know what asynchronous programming is and how to create our own asynchronous functions so now is a good time to come back to this chapter and study again how the futures objects work. In this exercise we're going to dive into these concepts again, including abstract classes, libraries, and asynchronous programming. We'll use Streams not just Futures, create our own asynchronous functions, and use two different libraries. We're going to create a new Web application project called `chapter_23_sample` and we'll create a `contacts.dart` file inside the `lib/` directory. This will be our main abstract class called `Contacts`. This is the same class we used in the Chapter 21 example.

```
library chapter_23_sample.idb_contacts;

import 'dart:async';
import 'dart:html';
import 'package:lawndart/lawndart.dart';
import 'package:google_maps/google_maps.dart';

/// Abstract class to manage contacts.
abstract class Contacts {
  /// Creates a new contact.
  /// It'll return the new contact ID.
  Future add(Map data);

  /// Updates an existing contact.
  Future update(var id, Map data);

  /// Deletes an existing contact.
  Future delete(var id);

  /// Gets data for a given contact id.
  Future get(var id);

  /// Lists the existing contacts.
  Future list();
}
```

```

    /// Search for contacts,
    Future search(String query);

    /// Gets map from contact location.
    Future map(String address, String container);
}

```

Note that in this class we've added a new method called `map` to get the map from the contact address location. We'll use Google Maps to draw the map on our application. We've created a new class to manage our contacts that extends from `Contacts` abstract class but in this case we're going to use `IndexedDb` on our browser to store the contacts and all of their information. This is our `IdbContacts` class using `IndexedDb` through the `lawndart` pub package and Google maps through the `google_maps` pub package. To use these packages you must add them to your `pubspec.yaml` and import them on your class.

```

import 'dart:async';
import 'dart:html';
import 'dart:math';
import 'package:lawndart/lawndart.dart';
import 'package:google_maps/google_maps.dart';

```

As you can see our `IdbContacts` class will use `dart:async` and `dart:html` packages. Below you can see the `IdbContacts` class with its methods, instance variables, and constructors.

```

/// Concrete class to manage contacts stored on browser using IndexedDb.
/// This class requires lawndart package.
/// https://pub.dartlang.org/packages/lawndart
class IdbContacts extends Contacts {

    /// IndexedDb database object.
    IndexedDbStore _db;
    /// Database name.
    String _dbName;
    /// Database table name.
    String _tableName;

    /// Constructor.
    IdbContacts([String dbName='idbContacts',
                String tableName='contacts']) {
        _dbName = dbName;
        _tableName = tableName;
        _db = new IndexedDbStore(_dbName, _tableName);
    }

    /// Delete all the information on the DB
    Future dropDB() => _db.open().then((_) => _db.nuke());
}

```

```

/// Creates a new contact.
Future add(Map data) {
  var now = new DateTime.now().millisecondsSinceEpoch;
  var rand = new Random().nextDouble().toString().split('.')[1].substring(0, 10);
  var id = 'c_$now$rand';
  return _db.open().then((_) => _db.save(data, id));
}

/// Updates an existing contact.
Future update(String id, Map data) {
  return _db.open().then((_) => _db.save(data, id));
}

/// Deletes an existing contact.
Future delete(String id) {
  return _db.open().then((_) => _db.removeByKey(id));
}

/// Gets data for a given contact.
Future get(String id) {
  return _db.open().then((_) => _db.getByKey(id));
}

/// Lists the existing contacts.
Future<Map> list() {
  var results = {};
  var c = new Completer();
  _db.open().then((_) {
    _db.keys().listen((key) {
      results[key] = _db.getByKey(key);
    }, onDone: () => Future.wait(results.values).then((vals) {
      var i = 0;
      results.forEach((k, v) {
        results[k] = vals[i];
        i++;
      });
      c.complete(results);
    }));
  });
  return c.future;
}

/// Search for contacts.
Future<Map> search(String query) {
  var matches = {};
  var c = new Completer();
  list().then((results) {
    results.forEach((k, v) {

```

```

        v.values.forEach((f) {
            if(f.toString().toLowerCase().contains(query.toLowerCase())) {
                matches[k] = v;
            }
        });
    });
    c.complete(matches);
});
return c.future;
}

Future map(String address, String container) {
    var c = new Completer();
    var req = new GeocoderRequest();
    req.address = address;
    new Geocoder().geocode(req, (results, status) {
        // Get lat, long for the address geocoder
        var latlng = results[0].geometry.location;
        final coords = new LatLng(latlng.lat, latlng.lng);
        // Map
        final mapOptions = new MapOptions()
            ..zoom = 18
            ..center = coords
            ..mapTypeId = MapTypeId.ROADMAP;
        final map = new GMap(querySelector(container), mapOptions);
        // Marker
        final markerOptions = new MarkerOptions()
            ..position = coords
            ..map = map
            ..title = address;
        final marker = new Marker(markerOptions);
        // complete the future.
        c.complete(true);
    });
    return c.future;
}
}

```

Let's see how to use this awesome class and how the future objects work here.

```

import 'package:chapter_23_sample/contacts.dart';

void main() {

    // Managing contacts on IndexedDb storage.
    var id;
    var idb = new IdbContacts();

```

```

// Adding new contact.
idb.add({'fname': 'Moises'}).then((k) {
  // new key created for this contact.
  id = k;
  print(id);
  // getting info for the given ID
  return idb.get(id);
}).then((contact) {
  print(contact);
  return idb.update(id, {'fname': 'Moisés', 'lname': 'Belchín'});
}).then((_) {
  print('updated');
  // showing changes after updating.
  return idb.get(id);
}).then((contact) {
  print(contact);
  // After updating we gonna delete the contact.
  return idb.delete(id);
}).then((_) {
  // getting a nonexistent key will return NULL
  return idb.get(id);
}).then((contact) => print(contact));
}

```

We've imported our `idbContacts` class located in the `contacts.dart` file. After we instantiate our `IdbContacts` class we've added a new contact to our browser database, when the `add` method finishes its job we'll get the new id for the contact added. With this id we can get its information using the `get` method.

We've updated the contact information by passing the id and a map with the information to update on our browser IndexedDb database. Finally we've deleted the contact using `delete` method. After running this code you will see these messages on your Output view window.

```

c_14180566623240908087689
{fname: Moises}
updated
{fname: Moisés, lname: Belchín}
null

```

Let's see now how to add multiple contacts, view it on our browser, and use the `list` and `search` methods to list all the contacts or to look for one of them.

```

import 'dart:async';
import 'package:chapter_23_sample/contacts.dart';

```

```

void main() {

  var idb = new IdbContacts();
  // Cleanup the database and add contacts to DB for list and search.
  Future.wait([
    idb.dropDB(),
    idb.add({
      'fname': 'Moises',
      'lname': 'Belchin',
      'address': 'paseo del prado, 28',
      'zip': '28014',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    idb.add({
      'fname': 'Patricia',
      'lname': 'Juberias',
      'address': 'Paseo de la Castellana, 145',
      'zip': '28046',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    idb.add({
      'fname': 'Peter',
      'lname': 'Smith',
      'address': 'Cyphress avenue',
      'zip': '11217',
      'city': 'Brooklyn',
      'country': 'EEUU'
    })
  ]).then((_) {

    // List all records
    idb.list().then((results) {
      results.forEach((k, v) {
        print('$k :: ${v['fname']}');
      });
    });
  });
}

```

If you run this code you will get these messages on your Output view window.

```

c_14180569406757565057583 :: Moises
c_14180569406785295107053 :: Patricia
c_14180569406785694021442 :: Peter

```

Now we'll show you how to see these new contacts on your Chromium browser. Open the developer tools on Chromium and go to the **Resources** tab. On the right pane you can see the different storages that Chromium supports. Go to **IndexedDB ► idbContacts ► contacts**. Here you can see the new contacts we added to IndexedDb.

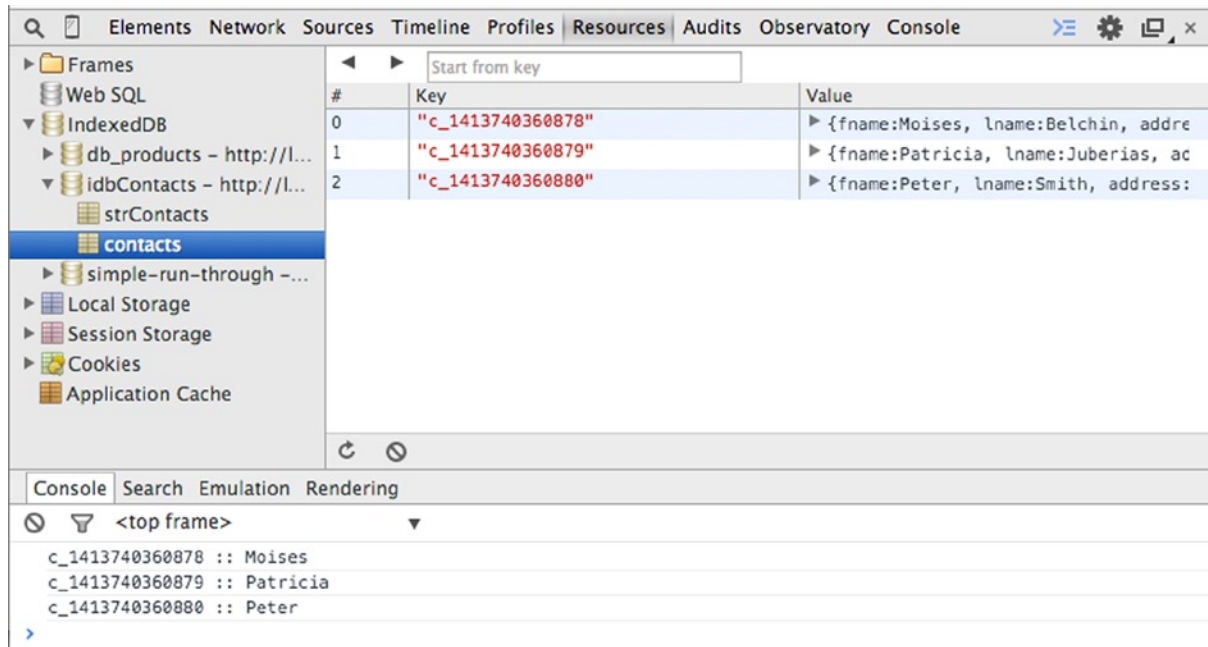


Figure 23-1. Viewing our new contacts on IndexedDb through Chromium Developer Tools

Let's see how to find a contact on our IndexedDb database using the `idbContacts` search method.

```

import 'dart:async';
import 'package:chapter_23_sample/contacts.dart';

void main() {

  var idb = new IdbContacts();

  // Cleanup the database and add contacts to DB for list and search.
  Future.wait([
    idb.dropDB(),
    idb.add({
      'fname': 'Moises',
      'lname': 'Belchin',
      'address': 'paseo del prado, 28',
      'zip': '28014',
      'city': 'Madrid',
      'country': 'Spain'
    }),
    idbC.add({
      'fname': 'Patricia',
      'lname': 'Juberias',

```

```

        'address': 'Paseo de la Castellana, 145',
        'zip': '28046',
        'city': 'Madrid',
        'country': 'Spain'
    )),
    idbC.add({
        'fname': 'Peter',
        'lname': 'Smith',
        'address': 'Cyphress avenue',
        'zip': '11217',
        'city': 'Brooklyn',
        'country': 'EEUU'
    )),
  ]).then((_) {

    // full search
    var term1 = 'cyphress';
    idb.search(term1).then((regs) {
      print('Searching for: $term1');
      if(regs.length <= 0) {
        print('No matches found');
      } else {
        print('Found ${regs.length} reg(s)'); // Found 1 reg(s)
        regs.forEach((k,v) {
          print('$k :: ${v['fname']}');
        });
      }
    });

    var term2 = 'Spain';
    idb.search(term2).then((regs) {
      print('Searching for: $term2');
      if(regs.length <= 0) {
        print('No matches found');
      } else {
        print('Found ${regs.length} reg(s)'); // Found 2 reg(s)
        regs.forEach((k,v) {
          print('$k :: ${v['fname']}');
        });
      }
    });
  });
}

```


You can see something like this after running this code sample on your Dart Editor.

```
Searching for: cyphress
Found 1 reg(s)
c_14180570189567842745959 :: Peter

Searching for: Spain
Found 2 reg(s)
c_14180570189524856050621 :: Moises
c_14180570189566459398852 :: Patricia
```

Finally we're going to show you how to use the `map` method to get the map from google maps using `google_maps` library and show it on our HTML web page. You must add a script tag to your `chapter_23_sample.html` file. Replace these tags located between `<head>` and `</head>`.

```
<script async type="application/dart" src="chapter_23_sample.dart"></script>
<script async src="packages/browser/dart.js"></script>
```

For these tags:

```
<script async type="application/dart" src="chapter_23_sample.dart"></script>
<script src="http://maps.googleapis.com/maps/api/js?sensor=false"></script>
<script async src="packages/browser/dart.js"></script>
```

Note that you'll need to select a key from those in your own `indexdb`.

```
import 'package:chapter_23_sample/contacts.dart';
void main() {

  var idb = new IdbContacts();
  idb.get('c_14180572994103464119734').then((contact) {
    var address = "" + "${contact['address']}, ${contact['zip']},
    ${contact['city']}, ${contact['country']}" + "";
    idb.map(address, '#sample_container_id');
  });
}
```

If you run this code on Dart Editor you can see after few seconds the map for the contact location.

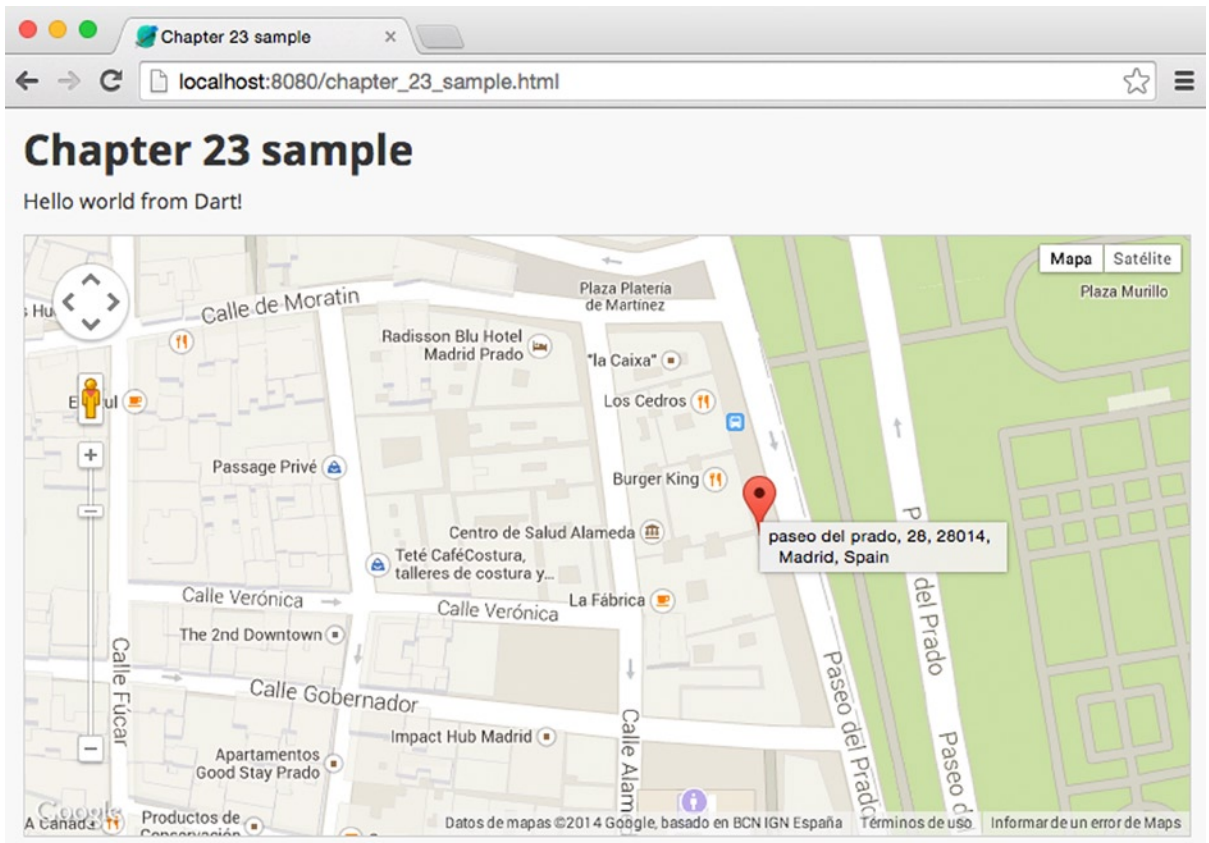


Figure 23-2. Loading the google map for the given contact address location

Summary

In this chapter we have learned:

- What asynchronous programming is
- What Future and Stream classes are
- How to work with Future objects
- How to work with Stream objects
- How you can create your own asynchronous functions in Dart using Future API constructors or Completers.
- How to combine abstract classes, libraries, packages, and your own asynchronous functions to develop an application for managing contact information stored in IndexedDb.



An Advanced Syntax Summary

When you learn a new programming language, it is very useful to have a cheat sheet to check quickly the main instructions and the syntax. Here's the summary of syntax we have learned until now.

Exceptions and Errors

Table 24-1 lists the exception management cheat sheet and some examples of use. In Tables 24-2 and 24-3 you can see the Exception and Error types supported in Dart.

Table 24-1. *Exception Management Cheat Sheet*

EXCEPTION MANAGEMENT	
throw	To throw exceptions
try {}, on {}, catch {}	To catch exceptions
finally {}	To execute a code snippet always, occurs or not an exception

Table 24-2. *Exception Types Cheat Sheet*

EXCEPTION TYPES	
Exception	Abstract class that represents a system failure. You must inherit from this class to create your own exceptions.
FormatException	Format error when a string or other value is not formatted correctly.
IntegerDivisionByZeroException	Except for dividing a value by zero.

Table 24-3. *Error Types Cheat Sheet*

ERROR TYPES	
AssertionError	Failure in an assert statement.
TypeError	Failure in a type assert statement.
CastError	Failed to convert a value from one type to another.
NullThrownError	When null value is thrown.
ArgumentError	An error occurs in function arguments.
RangeError	Try to access an index that is outside the object indexes range.
FallThroughError	Reached the end of a switch case.
AbstractClassInstantiationError	Try to instantiate an abstract class.
NoSuchMethodError	Try to access a nonexistent method on an object.
UnsupportedError	Operation is not allowed by the object.
UnimplementedError	The method is not implemented in the object.
StateError	Operation is not allowed by the current state of the object.
ConcurrentModificationError	Collections are modified during iteration.
OutOfMemoryError	Without enough memory.
StackOverflowError	Stack Overflow.
CyclicInitializationError	Lazily initialized variable cannot be initialized.

Let's see some examples of exception management. Below you can see how to throw exceptions or errors:

```
throw new Exception('The password is incorrect !!');
```

```
throw new TypeError();
```

Use try-on-catch to capture and handle exceptions.

```
try {
    login();
} on PasswordEmptyException { // Catches a specific exception.
    reLogin();
} on Exception catch(e) {    // Catches any type of exception.
    print('What happend here? ');
} catch(e) {                  // Catches anything else than will happen.
    print(' Something has gone wrong. I have no idea what happened !');
}
```

Use finally to execute code at the end of the try-catch block.

```

var myresult;
try {
    // I try to to access to the system.
    myresult = login();
} on Exception catch(e) {
    // An error occurs in the system.
    myresult = ' You did not say the magic word !';
} finally {
    // Displays the answer to the user.
    print(myresult);
}

```

Classes

In Tables 24-4 and 24-5 you can see the basic syntax for classes and inheritance, interfaces, abstract classes, mixins, generics, and typedefs.

Table 24-4. *Classes Cheat Sheet*

CLASSES	
class ClassName {}	Create a class.
var obj = new classConstructor();	Instantiates a class.
.method()	Access your instance or static variables or methods
.instance_variable	
this.method()	Self-reference
this.variable	
const ClassName {}	Constant class
_variableName	Private variable
< + [] > /	Operators you can override in your classes definition.
[] = <= ~/ & ~	
>= ^ * << ==	<pre> class GPS { num latitude; num longitude; GPS(this.latitude, this.longitude); GPS operator +(GPS g); GPS operator -(GPS g); } </pre>
- % >>	

Table 24-5. *Inheritance Cheat Sheet*

INHERITANCE	
class subClass extends parentClass	Create subclasses
super	Refer to Parent class
STATIC VARIABLES AND METHODS	
static method() {} static variable = value;	Create static methods or variables
INTERFACES	
class NewClass implements classA, classB, classC {}	Define a class that implements another class or classes.
ABSTRACT CLASSES	
abstract class myAbsClass {}	Abstract class defining
MIXINS	
abstract class myMixin {}	Class must extends from Object Class has no declared constructors Class has no calls to super You can add more functionalities into your classes using mixins without inheritance using with keyword. class MyClass extends Object with myMixin {}
GENERIC	
<...>	Indicate that the contents of a list, map, or your own classes is Generic. It would work with any generic value, num, string, etc.
TYPEDEFS	
typedef DataType FunctionName();	Indicates the data type result that this function should return.

Table 24-6 lists all the basic syntax to work with libraries.

Table 24-6. *Libraries Cheat Sheet*

LIBRARIES	
library LibName	Defines a Library
import 'LibName.dart'	Imports a Library
import 'dart:LibName'	Imports Dart native Library
import 'package:LibName.dart'	Imports a Package
import 'package:path/LibName.dart'	

(continued)

Table 24-6. *(continued)*

LIBRARIES	
<code>import 'LibName' as Prefix</code> <code>Ej. import 'dart:math' as Math</code>	Imports a Library with prefix
<code>import 'LibName' show partToShow</code>	Imports only a part of the library
<code>import 'LibName' hide partToHide</code>	Imports all names except <code>partToHide</code>
<code>library LibName;</code> <code>part 'file1.dart';</code> <code>part 'file2.dart';</code> <code>file1.dart:</code> <code>part of LibName;</code> <code>file2.dart:</code> <code>part of LibName;</code>	Creates libraries with multiple files In <code>file1.dart</code> and <code>file2.dart</code> you must indicate that these files are part of the library <code>LibName</code> with <code>part</code> of keyword.
<code>export LibName</code>	Exports a Library
<code>export LibName show partToExport</code>	Exports only a part of the library

In Table 24-7 you can see basic methods to work with Future and Streams.

Table 24-7. *Future and Stream Cheat Sheet*

FUTURE	
<code>Future fut = new Future(myFunction)</code>	Creates a future containing the result of calling your function asynchronously with <code>run</code> .
<code>Future.then()</code>	Get result of the asynchronous function
<code>Future.catchError()</code>	Get error of asynchronous function
<code>Future.whenComplete()</code>	Get the value when the asynchronous function has completed
<code>Future.timeout(Duration, onTimeout: () {})</code>	Detects excessive time consumed by asynchronous function and execute a given function.
<code>new Future.delayed(Duration duration, myFunction)</code>	Creates a future that runs your function after a delay.
<code>new Future.value("my_values")</code>	A future whose value is available in the next event-loop iteration.
<code>Future.wait([myFuture1, myFuture2]);</code>	Wait for all the given futures to complete and collect their values.
STREAM	
<code>Stream.listen(onData, onError, onDone, cancelOnError)</code>	Subscribe to the Stream events

PART V



Dart and Other Web Technologies