The Dart Programming Language

Gilad Bracha

The Dart Programming Language

However, Dart does not uphold the principle of uniform reference, due to the influence of Javascript and the C style syntax.

Mixins originated as an idiom in certain Lisp dialects. The linguistic model of mixins used here was introduced by William Cook and the author in 1990[4], and first implemented in Strongtalk[5],[6], an innovative Smalltalk system with a particularly strong influence on Dart. Closely related constructs exist in languages such as Newspeak, Scala (where they are called traits) and Ruby.

The name "expression problem" is due to Philip Wadler. The expression problem has been discussed extensively in the literature ([7], [8], [9]). Proposed solutions vary depending on the language and requirements involved.

2.17 Summary

Dart is a pure object-oriented, class-based language, which means that all runtime values are objects and every object is an instance of some class.

Objects have state and behavior. The state is only available via special accessor methods—getters and setters. This ensures that all computation on Dart objects is done via a procedural interface.

Classes are reified at runtime and so must be objects themselves. Hence every class is an instance of a metaclass of type `Type`. Every class has at least one constructor; constructors are used to create objects. Some objects are constant, meaning they can be pre-computed at compile-time.

Every Dart class has a unique superclass, except for the root of the class hierarchy, `Object`. All Dart objects inherit common behavior from `Object`.

Dart supports mixin-based inheritance: every class induces a mixin capturing its unique contribution to the class hierarchy. Mixins allow the code of a class to be reused in a modular fashion independent of its place in the class hierarchy.

Chapter 3

Libraries

Dart programs are organized into modular units called *libraries*. We saw our first library in Chapter 1. You might imagine this was the points library shown at the end of that chapter, but in fact, it was our "Hello World" program. A trivial library perhaps, but a library all the same. Here it is again:

```
main(){
  print('Hello World');
}
```

As you can see, there is no requirement for an explicit library declaration like the one we saw in points. Most libraries do have such a declaration, which is useful for various reasons we shall illustrate below. However, for quick, simple tasks, it is convenient to be able to just write a function and run it. In Dart we always seek to enable experimentation with the language with a minimum of fuss, while still providing support for good software engineering practice.

3.1 The Top Level

In our example, the library consists of a single top-level function main(). In general, a library consists of a number of *top-level declarations*. These declarations may define functions, variables and types.

Here is a slightly larger, but still trivial example—a library that implements a stack. It's the kind of dreary exercise you might see in an introductory programming class, which is good, because we can focus on the language constructs rather than on sophisticated program logic.

```
library stack1;
final _contents = [];
get isEmpty => _contents.isEmpty;
get top => isEmpty ? throw 'Cannot get top of empty stack' : _contents.last;
get pop => isEmpty? throw 'Cannot pop empty stack' : _contents.removeLast();
push(e) {
  _contents.add(e);
  return e;
}
```

We have one top-level variable, _contents, which is initialized to an empty list. Top-level variables induce implicit accessors, much as instance and class fields do. Again, no user code accesses the variables directly. Top-level variables are initialized lazily, the first time their getter is called, just like class variables. In stack1, _contents won't be set to [] until one of the methods that access it is called.

Together, top-level and class variables constitute the category of static variables. The two differ in their scope—where they may be referenced by name. Whereas the names of class variables are restricted to the scope of the class that declared them (not even subclasses see them), the names of top-level variables (also known as *library variables*) are in scope throughout the library that declares them. The library scope typically comprises multiple classes and functions.

As with class variables, top-level variables may be final, in which case no setter is defined for them and they must be initialized at their point of declaration. It is also possible to declare a static variable (be it a class or library variable) as a constant, in which case it can only be assigned a compile-time constant and is treated as a constant (Sections 2.11, 6.1.4).

The scope rules for top-level functions (often referred to as *library methods*) are the same as for top-level variables; a top-level function is in scope throughout the library. One can define regular functions, getters and setters. In each case, the body of the function can be given in the short form using => followed by a single expression (as in isEmpty, top, pop), or by a sequence of statements between curly braces (as in push()).

In addition to top-level functions and variables, we also have top-level class declarations. All class declarations in Dart are top-level since Dart does not support nested classes.

3.2 Scripts

The "Hello World" program is an example of a *script*, which is a directly executable Dart library. A script begins executing at its main() function. If a library doesn't have a main() function, it is, by definition, not a script and it cannot be run on its own.

Scripts have one further peculiarity. The first line of a script may start with the character # that can be followed by arbitrary text up to the end of the line. This is useful in many environments which are able to run various interpreters on files, based on a # prefixed directive at the head of the file to be interpreted.

3.3 Privacy

Libraries are Dart's unit of encapsulation. Names that begin with an underscore (_) are private to a library. We saw an example above, the library variable _contents. Making _contents private helps maintain the integrity of the stack abstraction introduced by stack1. Because only the code inside stack1 can access _contents, we are assured that no other code will tamper with the underlying representation of the stack.

As another example, consider

```
class CachingClass {
  var _cache;
  operator [](i) {
    if (_cache[i] == null) {_cache[i]= complicatedFunction(i);}
    return _cache[i];
  }
}
```

No one outside the library that defines CachingClass can get at the _cache field.

This scheme allows you (and the compiler or any other tool) to recognize whether something is private without having to look up its declaration.

Privacy should not be confused with security. The purpose of privacy in Dart is to support software engineering needs, not security needs. The only security boundaries are those between isolates, which we will examine in detail in Chapter 8. Within an isolate, there are no security guarantees.

3.4 Imports

Readers are no doubt familiar with the concept of imports in programming. For example, if we wanted to make use of stack1 in an application, we might write

```
import 'stack1.dart';
main() {
 push('gently');
 push('harder');
 print(pop);
 print(top);
}
```

providing our main() function with access to the push() and pop methods of stack1. The script will print harder and then print gently. This code will work provided that the library stack1 is stored in a file named stack1.dart in the same directory as our main script. However, what if stack1 was stored elsewhere, say somewhere on the web such http://staxRUs/stack1.dart?

We could replace the import with:

```
import 'http://staxRUs/stack1.dart';
```

Dart imports work with arbitrary URIs (Universal Resource Indicators). However, neither of the URIs above is particularly advisable because it makes your code sensitive to any change in the imported library's location. Using such URIs is useful for quick and dirty experiments, but serious code requires more discipline. Typically, one would write

```
import 'package:stack1.dart';
```

The schema package: is used to invoke a resident *package manager* that encapsulates the knowledge of where code resides. Dart environments typically come with a package manager, but the details are outside the scope of this book.

There is no need to use the package: schema for libraries that are part of the Dart platform. These are accessed using the dart: schema, as in:

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

Other examples include dart:html, dart:json and a good many others.

Whatever schema you use, the URI had better refer to an actual library, or a compiler error will occur. The URI also needs to be a constant string literal (6.1.1.4), and may not involve interpolation.

The names available within a library are those names it declares and those names introduced into the library scope via imports of other libraries. The names defined by dart:core are imported implicitly.

The set of names a library makes available to its clients differs from the names available internally. First, the names a library imports are not transitively made available to clients that import the library. Also, private members of a library are not available to other libraries that import them. There are additional differences we shall see shortly. It is therefore useful to speak of the *exported namespace* of a library.

Suppose we had an alternate stack implementation in a library stack2. We might write some code designed to test the two implementations. We might start by sketching out our script like this:

```
import 'package:stack1.dart';
import 'package:stack2.dart';
main() {}
```

So far, so good. You can compile this code without any issues. However, once we try to fill out the body of main() with code that uses the imports, we will hit a snag.

```
import 'package:stack1.dart';
import 'package:stack2.dart';
main() {
 // testing stack1
  push('gently'); // Static warning
  push('harder'); // Static warning
  print(pop); // Static warning
  print(top);  // Static warning
  // testing stack2
  push('gently'); // Static warning
  push('harder'); // Static warning
  print(pop); // Static warning
  print(top); // Static warning
}
```

Can you tell the difference between the code that uses stack1 and the code that uses stack2? Of course not, and neither can the compiler. The methods of stack2 have the same names as those of stack1, and importing them both into the same scope is hopelessly ambiguous.

The Dart compiler will issue a warning about every use of the ambiguous names. Notice that importing conflicting names does not in itself cause any warning—warnings are issued only if you try to use an ambiguous name. This is in keeping with Dart's "keep out of the way" philosophy. It also has the nice property that when someone adds a top-level name to a library you imported earlier, your code is less likely to break.

If we ignore the warnings and try to run the code, the first call to push() will result in a runtime error. Specifically, a NoSuchMethodError will be raised, because push() is not well defined. Readers might well ask why is the code even allowed to run? We know with certainty that it will fail, so why not mark these situations as errors and refuse to compile the program?

Bear in mind that the use of an ambiguous name might be in a branch that is not always executed. Should we prevent a developer from testing another branch until the ambiguity is eliminated? We think not; we do not want to impose a specific workflow on the programmer. Instead, we are content to inform programmers of the problem, and leave it to them to decide when to resolve it. In general, Dart seeks to avoid compilation errors as much as it can in order to avoid imposing a specific ordering of tasks on the programmer.

A good way to overcome the kind of ambiguity shown above is to distinguish the two imports by providing each set of imported names with a distinct prefix.

```
import 'package:stack1.dart' as stack1;
import 'package:stack2.dart' as stack2;
main() {
 // testing stack1
  stack1.push('gently');
  stack1.push('harder');
  print(stack1.pop);
  print(stack1.top);
  // testing stack2
  stack2.push('gently');
  stack2.push('harder');
  print(stack2.pop);
  print(stack2.top);
}
```

Dart gives priority to the declarations within a library over any imports, so adding a top-level name to an imported library is not as damaging as it might be. Problems can still arise, however. Additions to an imported library can cause problems in your code if you access a name from one import, and another imported library later adds the same name. In addition, a newly imported name can shadow an inherited name.

Prefixing all imports ensures that when imported libraries add top-level members, they can never cause breakage to the importer in the future.