DZone Refcardz

CONTENTS INCLUDE:

Core Dart

The Next-Generation Client-Side Web Programming Language

By Chris Buckett

ABOUT DART

Dart is a dynamic, optionally typed, object oriented, class based language supporting single inheritance, multiple interfaces, and functional programming. All Dart apps start with a main() function, and Dart code looks like this:

```
main() {
  var greeting = "Hello"; // dynamically typed
  String audience = "World"; // type annotation
  sayGreeting("$greeting $audience"); // call function
}

// shorthand (single-line) function syntax
sayGreeting(message) => print(greeting); // "Hello World"
```

A language for the web

Dart is designed as a web language that can be compiled to JavaScript or run in a browser-embedded virtual machine. The JavaScript that is output is targeted at modern web browsers, which makes Dart a perfect match for HTML5. Dart can also run on a server-side version of the virtual machine, using evented I/O to interact with the host operating system (Windows / Mac / Linux).

To run the previous script in a Dart-enabled web browser, you can embed it in an HTML script tag, with a mime-type of application/dart, as shown below:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <script type="application/dart" src="hello.dart">
    </script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

The dart.js script is a JavaScript bootstrapper that detects whether the Dart VM is embedded in the browser. The "Hello World" message is output to the browser's debug console.

OPTIONAL TYPING EXPLAINED

Dart's types are often called type annotations. This is because they have no effect on the running of the code in production; instead, they provide information to tools and other developers about the coder's intention. The following function is written twice, first with no type annotations, and then with type annotations added:

```
// no return type or parameter type annotations defined
repeatGreeting1(greeting,count) {
  var result = []; // empty list literal
  for (var i = 0; i < count; i++) {
    result.add(greeting);
  }
  return result;
}

// return type, parameter types and variable types
// declared, and highlighted in bold font
List repeatGreeting2(String greeting, int count) {
  List result = new List(); // list constructor
  for (int i = 0; i < count; i++) {
    result.add(greeting);
  }
  return result;
}

main() {            ------Snippet cont'd on next column--------->
  // identical function calls
  print(repeatGreeting1("Hello", 3));
  print(repeatGreeting2("Goodbye", 3));
}
```

If these two functions run identically, then why have types at all? repeatGreeting2() declares that the greeting parameter should be a String type. What happens if, as follows, we pass an integer value 9999 in place of a string?

```
main() {
  print(repeatGreeting1(9999, 3));
  print(repeatGreeting2(9999, 4)); // type warning
}
```

Dart will not prevent this code from running, because type annotations are ignored at runtime. The Dart tools, however, will report a type warning on the call to repeatGreeting2(), as you are trying to pass int when it expects a String.

Checked Mode vs Production Mode

Although Dart will not prevent this code from running, when you run this code through the Dart Editor, a special "Checked Mode" is enabled in the Dart VM that attempts to halt execution when it encounters these types of warnings. During production mode (when you convert your Dart code to JavaScript, run in the browser without using the editor, or run a server-side app on the server-side VM), checked mode is switched off.

Hot Tip You can enable checked mode in the stand-alone Dart VM by passing in the parameter --checked from the command line.

DART TOOLS

The Dart language is made to be toolable, and as such, its tool ecosystem uses optional types to validate your application's source code to trap errors early.

The Dart tool ecosystem encompasses the Dart Editor and "Dartium", a custom build of Chromium with the Dart virtual machine embedded. This pair of tools supports features such as integrated debugging, code auto-completions, and refactoring. The Dart Editor is also available as an Eclipse plugin from http://www.dartlang.org/eclipse/update.

Behind the scenes, the Dart Editor uses the dart_analyzer tool, a static analyzer that will quickly find type errors in your code. You can integrate the dart_analyzer into your continuous integration system to trap errors when your code is committed to source control. The Dart2js tool compiles Dart source code to JavaScript, "treeshaking" to remove dead code as it goes, while Dartdoc outputs navigable HTML API documentation from your code comments. Finally pub, Dart's package manager tool, pulls in library dependencies from the web. These are all available in the Dart Editor menus, as stand-alone command-line tools in the dart-sdk/bin folder.

Using Pub

When you use the Dart Editor to create a new project, you are supplied with a pubspec.yaml file, which you use to provide configuration and dependency information to your project, and a starting-point project layout. By convention, following files and folders are used:

File / Folder	Description
<project_root>/	Main folder containing the project. Add this to source control.
pubspec.yaml	Dependency information and project configuration
README.md	Markdown formatted readme file
LICENSE	License information
lib/ awesome.dart src/ functions.dart classes.dart	The lib folder contains your library files, and any other source files (in a src subfolder) that make up your library.
web/ index.html index.css index.dart	(Web projects only) The web folder contains files that will be published to the web.
test/ awesome_test.dart	Executable test scripts to test your application.

In addition, the following files and folders may exist in your project folder:

File / Folder	Description
pubspec.lock	Contains version numbers of your project's dependencies. Useful when you want your entire team to be using the same version of dependencies (otherwise, don't commit to source control)
/packages/	Packages subfolders are symlinked throughout your project by the pub tool to enable relative access to all the library dependencies.
doc/	Include hand-authored documentation about your project here.
bin/	If your app is designed to be run from the command line, include the dart script to execute your app here.
example/	If examples are useful to users, place them here
tool/	Any helper scripts for internal use within the package go in the tool folder.

pubspec.yaml

The pubspec.yaml file provides configuration information about your project, and has a GUI built into the Dart Editor. The following Pub commands are available:

Summary of pub commands

Command	Description
pub install	Install new dependencies (runs automatically in the editor when you edit the pubspec file). Stores dependency versions in the pubspec.lock file.
pub update	Updates dependencies, pulling the newest versions allowed by the dependencies configuration (and overwriting those in the pubspec.lock) file.
pub publish	Publish your package to Google's pub repository at http://pub.dartlang.org

CORE TYPES

Dart is an object-oriented language with class-based inheritance similar to Java or C#. All types descend from the base Object class. The var keyword declares a variable and is a substitute for a strong type annotation. Variables must be declared before they are used (unlike JavaScript), and objects created by using the new keyword to invoke the object's constructor:

```
var myObject = new Object(); // declare myObject and
                             // construct the Object
print(myObject); // use the object
```

All objects inherit the `Object.hashCode()` and `Object.toString()` methods. The `toString()` method is called automatically whenever you try and convert an object into a string representation, such as when using the `print()` function:

```
// equivalent
print(123);
print(123.toString());
```

Hot Tip Stylistically using the var keyword for variable declaration should be preferred within code bodies. Only use explicit type annotations (such as String, int) at the "surface area" of your code, for example, in function parameter lists, return types, and class properties.

Dart has special support for a number of core, "primitive" types that are baked into the language. Although these types are objects, extending from the Object base class, you create instances of these objects literally, without using the new keyword.

num, int, double

`num` is the base numeric type, and has two implementations: `int` and `double`. num declares the basic operators of +-/* and functions such as `abs()`, `ceil()`, `floor()`, and `round()`, amongst others.

num

Use `num` to declare that you want a numeric value, whether an integer or floating-point.

```
num myNum = 123.45;
num myRounded = myNum.round();
```

int

The `int` type accepts integer values of arbitrary precision:

```
int myInt = 123;
// or
var myInt = 456;

var myIntAsHex = 0xABCDEF12345;
var myBigInt = 65498763215487654321654987654321354987;
```

double

The `double` type accepts 64-bit floating point numbers as specified in the IEEE 754 standard:

```
double myDouble = 123.45;
// or
var myDouble = 123.45;

var myDoubleWithExponent = 1.23e4;
```

bool

In Dart, only the literal boolean true is considered true. All other values are false. An extra restriction in checked mode only allows boolean operations on `true` and `false` literal values.

```
bool myTrue = true;
// or
var myTrue = true;

var myFalse = false;

print(myTrue == myFalse); // equals: false
print(myTrue != myFalse); // not equals: true
print(myTrue == myFalse || myTrue == myTrue); // or: true
print(myTrue && myTrue); // and: true
```

You can us the `is` and `is!` syntax to check a type's instance:

```
print("I am a string" is String); // prints "true"
print("I am not an int" is! int); // prints "true"
```

String

Dart strings are declared with either single or double quotes, and you can mix strings containing either:

```
String myString = "double quoted";
var myString = 'single quoted';
var myString = "contains 'single' quote";
var myString='contains "double" quote';
```

Multi-line strings are declared with three double-quotes:

```
var myString = """This is
a multi-line
string and can contain "double" and 'single' quotes""";
```

Strings can also contain escape codes, such as \n for new line:

```
var myString = "Line 1\nLine 2";
```

But you can also declare a raw string that will ignore escape codes by using the prefix `r`.

```
var myString = r"Line1\nStill line 1";
```

String interpolation

Dart strings cannot be concatenated using the + operator. The following code is not valid:

```
var myString = "hello" + " world"; // invalid syntax!
```

Instead, you must use string interpolation, which converts variables or expressions embedded within a string into a string. The following embeds two variables into a string by prefixing the variable name with $:

```
var greeting = "Hello";
var audience = "World";
var message = "$greeting $audience";
print(message); // Hello World
```

An expression enclosed within ${...} is evaluated before it is inserted into the string, for example:

```
var answer = "result=${1+2}";
print(answer); // result=3
```

The expression contained within curly braces can be any valid Dart expression, including function calls, for example:

```
var answer = "result=${1.23.round()}";
print(answer); // result=1.0
```

List

Dart doesn't have an explicit array type. Instead, it has a List type, which you can declare using either literal syntax, or as a List object. Internally, the object is the same.

```
// using list literal syntax
List myList = []; // empty list
// or
var myList = []; // empty list

var myList = [1,2,3,4]; // all int values
var myList = ["Item 1", "Item 2", 3, 4]; // mixed values
// using List object constructor
List myList = new List(); // empty list
// or
var myList = new List(); // empty list
var myList = new List(10); // list containing 10 nulls
var myList = new List.filled(10,"AA"); // contains 10
                                       // "AA" strings
```

You can add items to these lists by using the `add()` method, and access values using zero-based indexer `syntax [ ].` Use the length property to discover how many items are in the list:

```
var myList = [];
myList.add("item 1");
myList.add("item 2");
myList[0] = "ITEM 1"; // replaces "item 1";
print(myList[1]);  // "item 2";
print(myList.length); // 2
```

Lists can also be fixed size by using the `List.fixedLength()` constructor. Lists created in this way cannot have items added to them.

```
var myList = new List.fixedLength(10); // fixed length
myList.add("item11"); // throws UnsupportedError
// create a fixed length list where each element
// is filled with the string "AA"
var myList = new List.fixedLength(10,fill="AA";);
```

Using Generics

Lists allow use of generic types. This means that you can use type annotations to tell Dart and fellow developers that you are expecting your list to contain only certain types. For example, the following list should contain only Strings. Adding an integer would raise a type warning.

```
var myList = new List<String>();
myList.add(123); // warning: 123 is not a String
var myList = <String>["a","b"];
myList.add(123); // warning: 123 is not a String
```

Iterating Lists

You can iterate a list with a for loop, for-in, or forEach:

```
var myList = ['hello','world']; // create a list

// for loop similar to C# / Java
// accessing elements by index
for (var index=0; index < myList.length; index++) {
  var value = myList[index];
  myList[index] = value * value; // store the new value
}

// for-in loop iterates each item
for (var item in myList) {
  print(item);
}

// using a lists forEach function, passing
// in an anonymous function callback
myList.forEach( (item) { //
  print(item);           // anonymous function callback
} );
```

Other collections

Other collection classes include HashSet, Queue and Set, which offer more specialized functionality.

Map

Like Lists, you can create maps (comma-separated list of key:value pairs) using either a map literal or a class constructor:

```
// Map Literal syntax
Map myMap = {"key1":"value1", "key2":"value2"};
// or
var myMap = {"key1":"value1", "key2":"value2"};
var myMap = {}; // empty map

// Map constructor syntax
Map myMap = new Map(); // empty map
// or
var myMap = new Map(); // empty map
```

Using Generics
Any type can be used as a map's key or a value, and you can mix key and value types within a map. To restrict the key or value types, you can create maps using Generic typing (as with Lists)

```
// key is String, value is any Object
var myMap = <String,Object>{}; // empty map
var myMap = new Map<String,Object>();
```

You access and insert items into a map by using a key as an indexer. If the key does not already exist, it will be replaced.

```
var myMap = new Map();
myMap["key1"] = "value1";
myMap["key2"] = "value2";
print(myMap["key2"]); // value2
```

Accessing a key that has not been added will return null, but you can also explicitly check if the key exists using the `containsKey()` method.

```
var myMap = {"key1":"value1"};
print(myMap["key2"]); // null
print(myMap,containsKey["key2"]); // false
```

You can easily access a map's keys and values properties:

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

// access the keys
for (var key in myMap.keys) {
  print(key);
}

// access the values
for (var value in myMap.values) {
  print(value);
}
```

You can also iterate the key:value pairs using a forEach function callback:

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

myMap.forEach( (key, value) { //
  print("$key = $value");     // anonymous callback
} );
```

FUNCTIONS

Functions live in top-level scope, inside function scope, or within a class declaration (where they are known as methods).

Dart has both longhand and shorthand functions. Shorthand functions consist of a single expression, and always return the value of that expression, for example:

```
sayHello(name) => print("Hello $name"); // returns null
sum(num val1,num val2) => val1+val2; // returns result
```

Shorthand functions can never be declared to return `void`.

The long-hand equivalent of the functions declared above is shown below. They need to explicitly use the return keyword to `return` a value.

```
sayHello(name) {
  return print("Hello $name"); // returns null
}

sum(num val1, num val2) {
  return val1 + val2; // returns result of val1+val2
}
```

Longhand functions may or may not return a value. You can provide explicit return types on the function definition. The return type void declares that a function has no return type.

It is also good practice to declare type annotations on function parameter lists. This lets the Dart tools and other developers discover your intent.

```
void sayHello(String name) {
  print("Hello $name"); // no return value
}
num sum(num val1, num val2) {
  return val1 + val2;
}
```

Function Parameters

Functions can have both mandatory and optional parameters. Mandatory parameters are declared first and are always positional. Optional parameters are named or positional, but not both.

When you call a function, you must always provide values for the first set of mandatory parameters, but not the optional parameters.

Optional positional parameters must be provided in the order in which they are defined. Optional named parameters can be provided arbitrarily as {key1:value1,key2:value2} pairs.

Here are some examples – the function bodies are omitted:

```
/** definition with mandatory
 *  all params are mandatory
 */
void mandatoryParams(String name, int version, url) {
  // snip function body
}
/** definition with mandatory and
 *  optional positional parameters.
 *  [name] is mandatory, [version] and [url] are optional.
 *  [version] defaults to 1 if not supplied.
 *  [url] defaults to null if not supplied.
 */
void optionalParameters(String name,
                        [int version=1, url]) {
  // snip function body
}
/** definition with mandatory and
 * optional named parameters
 * [name] is mandatory, [version] and [url] can be
 * passed independently of each other
 */
void optionalNamed(String name, {int version:1, url}) {
 // snip function body
}

// calling, elsewhere in code:
// mandatory parameters passed in the correct order
mandatoryParams("Dart",1,"http://www.dartlang.org");

optionalParameters("Dart");
optionalParameters("Dart",1);
optionalParameters("Dart",1,"http://www.dartlang.org");

// optional named parameters in any order as
// key:value pairs, within braces { }
optionalNamed("Dart",{version:1});
optionalNamed("Dart",{url:"http://www.dartlang.org"});
optionalNamed("Dart",
          {version: 1, url:"http://www.dartlang.org"});
optionalNamed("Dart",
          {url:"http://www.dartlang.org", version: 1});
```

Within a function body, you can discover whether an optional parameter was supplied by using using the ?paramName syntax, for example:

```
void optionalNamed(String name [int version=1]) {
  if (?version) {
    // the caller explicitly supplied a version
  }
}
```

Function Variables

You can store a function in a variable just like any other type.

```
main() {
  // shorthand function stored in sayHello variable
  var sayHello = (name) => print("Hello $name");

  // longhand function in sayGoodbye variable
  var sayGoodbye = (name) {
    print("Goodbye $name");
  };

  // calling sayHello and sayGoodbye functions
  sayHello("Dart");
  sayGoodbye("Dart");
}
```

Because you can store a function in a variable, you can also pass a function into another function as a parameter. The following example passes a function as the third parameter to the getResult function, calling the returning the calcFunction's result. Note the use of Function as type information for the third parameter.

```
/**
 * Applies the function contained within calcFunction to
 * the values stored in val1 and val2
 */
num getResult(num val1, num val2, Function calcFunction) {
  return calcFunction(val1, val2);
}

main() {
  // declare two functions
  var add = (a,b) => a+b; // add a + b
  var subtract = (c,d) => c-d; // subtract d from c

  var result1 = getResult(1,2,add);
  print(result1); // 3

  var result2 = getResult(100,10,subtract);
  print(result2); // 90
}
```

Using Typedef

It is good practice to provide type annotations to your function's parameters, as we did with the Function type. You can be even more specific, though, and make the getResult() function's third parameter only accept a function with a defined function signature. You use the typedef keyword to define a custom function type:

```
// define a function signature:
typedef CalcType(num value1, num value2); // function type

// Third parameter accepts a CalcType of function
num getResult(num val1, num val2, CalcType calcFunction) {
  return calcFunction(val1, val2);
}

main() {
  // function signature matches CalcType
  var add = (num a,num b) => a+b;

  // function signature does not match CalcType
  var sayHello = (name) => print("Hello");
  var result1 = getResult(1,2,add); // works fine
  var result2 = getResult(1,2,sayHello); // type error
}
```

Closures

A closure is a construct that "closes-over" variables that are in its scope. This is best explained with an example:

```
getClosure() {
  var myVariable = "Hello"; // a variable in the scope
                            // of getClosure function

  var myFunction = () { // a function, also declared in
    print(myVariable);  // the scope of getClosure
  }                     // function, that uses myVariable
                        // (declared above)

  return myFunction;    // myFunction is returned by
                        // getClosure
}
```

When `getClosure()` returns `myFunction`, the variable called `myVariable` is no longer in `getClosure's` scope. The function defined by `myFunction`, however makes use of `myVariable`, so `myVariable` still exists within the scope of `myFunction`. You can use `getClosure` as follows:

```
main() {
  var aFunction = getClosure(); // myFunction is now
                                // stored in variable
                                // aFunction

  // call aFunction
  aFunction(); // prints "Hello";
}
```

CLASSES AND INTERFACES

Dart's class system is very similar to Java and C#. It is single-inheritance with multiple interfaces.

Defining Classes

An example class with a constructor, methods, fields and getters and setters is shown below. All constructors, methods, getters and setters can use either longhand or shorthand function syntax, and optional parameters.

```
class Language {
  String name; // public field
  int _version; // private field: _ underscore prefix denotes
                // private within the same library
  static url = "http://www.dartlang.org"; // same value
                                          // exists across
                                          // all instances
  final awesomeness = 11;  // immutable after
                           // constructor initialization

  // default constructor
  Dart() {
    name = "Dart";
    _version = 1;
  }

  // named constructor with initializer to initialize
  // final variables before class construction.
  Dart.withExtraAwe(aweValue) : awesomeness=aweValue {
    name = "Dart";
    _verison = 1;
  }

  // Getter and setter pair.  Getters and setters are
  // independent of each other, so you can have one
  // without the other
  int get version => _version; // shorthand syntax
                               // returns _version
  set version(int val) {  // longhand syntax
    _version = val; // sets the _version
  }

  // private method, shorthand syntax
  bool _isAwesome() => this.awesomeness > 10;

  // public method
  void checkAwesomeness() {
    if (_isAwesome) {
      print("$name $version is $awesomeness out of 10");
      // prints: "Dart 1 is 11 out of 10"
    }
  }
```

Using your class

You use the various features of your class as shown below:

```
main() {
  var lang = new Language(); // using default constructor
  // using named constructor
  var langPlus = new Language.withExtraAwe(12);

  // the syntax for accessing getters and setters or field
  // variables is identical.  This means you can use them
  // interchangeably
  lang.version = 1.1; // calling the setter
  lang.name = "#dartlang"; // setting a field

  print(lang.version); // calling a getter
  print(lang.name); // reading a field value

  lang.checkAwesomeness(); // calling a method

  // updating a static field happens on the object itself
  Language.url = "http://api.dartlang.org";
  print(Language.url); //
```

Class inheritance

Classes use the extends keyword to inherit another class. Dart uses the same object-oriented principals Java and C#. To access the parent object, use the super keyword. Constructors are not inherited, and super constructors must be called in the constructor initialization block.

```
class LanguageBinary extends Language {
  // calling the Language's withExtraAwe named constructor
  // in the LangaugeImplementation's constructor
  // initialization block
  LanguageImplementation(awe) : super.withExtraAwe(awe) {
    name = "DartVM"; // defined on Language
    version = 1; // defined on Language
    _executable = "dart.exe" // introduced in this
                             // child class
  }
  String _executable;
  get String executable => _executable;
}

  // final fields cannot be updated
  lang.awesomeness = 12; // error
 }
```

Abstract Classes

If you want to provide a class without any implementation, or with a partial implementation, use the abstract keyword. Classes that inherit the abstract class must provide the missing implementation.

```
abstract class Environment {
  String getOpSystem(); // no method body

  bool isWebBrowser() { // method body provided
    // snip some implementation
  }
}
```

Class interfaces

In Dart, every class is also an interface – there are no explicit interface definitions. A class's interface is the public representation of the class's members.

A class is said to implement the interface of another class, using the implements keyword. This means that it makes the same public members available as defined on the classes that are being implemented. Because the calling syntax for getter / setter pairs, and fields are interchangeable, either can be used.

Both abstract classes and regular classes can be used as interfaces, as shown below:

```
class BrowserScript implements Language, Environment {
  // implementing public interface of Language
  int version;
  String name;
  void checkAwesomeness() => print("Totally awesome");

  // implementing public interface of Environment
  bool isWebBrowser() => true;
  String getOpSystem() => print("ChromeOS");
}
```

When a class implements an interface, it must provide all the members. From Dart's point of view, the new class is now an instance of whatever it is implementing.

```
main() {
  var script = new BrowserScript();
  print(script is BrowserScript); // true
  print(script is Language); // true
  print(script is Environment); // true

  // valid function calls:
  checkLanguage(script);
  getOs(script);
}
```

ABOUT THE AUTHORS

Chris Buckett is a Technical Manager at Entity Group in the UK. He builds enterprise scale line of business web apps with GWT, Java and .Net, and provides solutions to enterprise data problems. Chris is a Google Developer Expert in Dart, author of Dart in Action, runs http://blog.dartwatch.com, and curates the http://DartWeekly.com newsletter. He believes in using the best tool for the job, and Dart is a great tool for building modern browser web applications.

RECOMMENDED BOOK

Dart in Action introduces Google's Dart language and provides techniques and examples showing how to use it as a viable replacement for Java and JavaScript in browser-based desktop and mobile applications. It begins with a rapid overview of Dart language and tools, including features like interacting with the browser, optional typing, classes, libraries, and concurrency with isolates. After you master the core concepts, you'll move on to running Dart on the server and creating single page HTML5 web applications.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "'DZone is a developer's dream'," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-73-8
ISBN-10: 1-936502-73-9
50795
9 781936 502738

$7.95

Version 1.0