APPENDIX A

Dart Cheat Sheet

This appendix is by no means meant to be a comprehensive Dart reference. Nor is it an exhaustive review of every language feature covered in the book. Instead, it is meant to be a quick-reference guide to the core language and some of the most commonly used classes in the standard library. If you've read the whole book, you will have seen all of this material before. If the material of a particular section is covered wholly in a chapter or two, then those chapters are listed in parentheses after the topic. The purpose of this appendix is to quickly give you an example of how something may appear syntactically. It is not meant to explain "why." For that, you'll have to read the book! There is no new material in this appendix. All of the content has been covered in the main text in far greater depth.

The Basics

Dart's basic syntax is straightforward, consistent, and similar to that of other popular languages, such as C, JavaScript, Java, and C#.

Declaring and Initializing Variables (Chapter 3)

Variables are declared in Dart optionally using types.

```
var m;  // no type specified
int n;  // n is specified as an int
```

When they are declared, variables can also be initialized.

```
var m = 4;
int n = 5;
```

Multiple variables of the same type can be declared at once.

```
int n = 5, o = 88, p;  // p is not initialized with a value
```

The new operator is used for initializing a new instance of a class. After a class is initialized, its constructor is immediately called. Parameters to the constructor are passed at initialization time.

```
Point p = new Point(24, 36);
```

www.it-ebooks.info

Literals (Chapter 3, Chapter 6)

Literals are used for creating objects, based on specific predetermined values. Literal syntax exists for Dart's core built-in types (see Table A-1).

Table A-1. *Fundamental Dart Types That Can Be Created with Literals*

Type	Description	Example Literals
`int`	Integers	`5, -20, 0`
`double`	Floating-Point Numbers	`3.14159, -3.2, 0.00`
`String`	Strings	`"hello", "g", "To be or not to be?"`
`bool`	Booleans	`true, false`
`List`	Lists (Chapter 6)	`[1,2,3], ["hi", "bye"]`
`Map`	Maps (Chapter 6)	`{"x": 5, "y":2}`

Common Operators

Dart's built-in operators provide the backbone of the language (see Tables A-2, A-3, and A-4). There are more operators than those covered here, but these are the ones that were examined in this book. Dart also allows operator overloading, in which operator behavior for a particular class is specified (see Chapter 12).

Table A-2. *Arithmetic Operators*

Operator	Description	Example

`++`	Increment integer by 1	`x++;`
		`++x;`

`--`	Decrement integer by 1	`x--;`
		`--x;`

`+=`	Increment integer by arbitrary amount	`x += 34;  // increment x by 34`
`*`	Multiplication	`x = y * 5;`
`/`	Division	`x = 6 / 3;`
`~/`	Integer Division	`x = 6 ~/ 4;  // x is 1`
`+`	Addition	`x = y + 2;`
`-`	Subtraction	`x = 20 - 4;`
`%`	Remainder (modulo)	`x = 6 % 4;  // x is 2`

www.it-ebooks.info

Table A-3. *Operators That Return Boolean Values*

Operator	Description	Example
`==`	Equal to	`if (x == y) {...}`
`!=`	Not equal to	`if (x != y) {...}`
`>`	Greater than	`if (x > y) {...}`
`<`	Less than	`if (x < y) {...}`
`>=`	Greater than or equal to	`if (x >=y) {...}`
`<=`	Less than or equal to	`if (x <= y) {...}`
`!`	Not	`if (!x) {...}`
`||`	Logical Or	`if (x || y) {...}`
`&&`	Logical And	`if (x && y) {...}`
`is`	Check the type of an object	`if (x is Element) {...}`

Table A-4. *Other Operators*

Operator	Description	Example
`.`	Access variables/properties and call methods on objects	`y = x.children;` `x.executeTimely();`
`..`	Cascade operator –lets you make multiple calls on the same object	`x = new Car()` `..resetOdometer()` `..paintRed()` `..oilLife = 100.0;`
`+`	String concatenation (when used between two String objects)	`x = "Hello" + "there!";  // x is "Hello there!"`
`[]`	Index into List objects or Map objects	`x[2] = 34;          // sets element 2 of a List, x, to 34` `y["gallons"] = 34;  // sets the value associated with the key "gallons" in a Map, y, to 34.`
`()`	Call function	`myFunction();`
`=`	Assignment operator	`x = 5;`
`as`	Cast an object from one type to another	`(x as ImageElement).src = "...";`
`?`	Conditional operator—if the statement is true then returns first alternative; otherwise, second	`hot = (temp > 75) ? true : false;`

www.it-ebooks.info

Control Structures (Chapter 3)

if-statements are used for branching, based on the validity of a particular condition.

```
if (j > 5) { // execute following lines if statement in () is true
...
} else if (j < 2) {  // if first statement was false and this statement is true then:
...
} else {  // executed only if all other statements were false
...
}
```

switch-statements are convenient when branching correctly would require many else-if-statements.

```
switch (x) {
  case 5:
    ...
    break;
  case 6:
    ...
    break;
  default:
    ...
    break;
}
```

The ? operator is similar in concept to a very short if-statement. If the statement before the ? evaluates to true, then the first statement after it is evaluated. Otherwise, the second statement after it (the two statements are divided by a :) is evaluated.

```
String casualPants = (temp > 75) ? "shorts" : "jeans";
```

Loops (Chapter 3, Chapter 6)

for-loops are probably the most commonly used loops. They are declared with a setup step; conditional step; and step to be completed on each loop iteration (for (*setup step*; *conditional step*; *iteration step*)). The setup step is run before the loop's first iteration. The conditional step is executed once at the beginning of each iteration. The last step is executed after each loop iteration. The loop continues to iterate until the conditional step is false.

```
for (int i = 0; i < 10; i++) {...}
for (Cell cell = new Cell(); cell.color != blue; cell.blink()) {...}
```

while-loops keep executing until a condition is no longer true. A do-while-loop is like a while-loop, except its condition is evaluated after each iteration, instead of before.

```
while (x < 5) {...}
do {...} while (x < 5);
```

www.it-ebooks.info

for-in-loops are used for going through the contents of an Iterable. All of the collection classes that come with the Dart SDK implement Iterable.

```
List j = [1, 5, 9];
for (int x in j) {
  print(x);  // will print 1 then 5 then 9
}
```

Numbers

int is the class of integers, and double is the class for floating-point numbers. Both int and double are subclasses of num. Both int and double provide static parse() methods for intepreting String objects as their respective type of numbers. They usually should be wrapped with exception handling code, in case something goes wrong, such as a FormatException.

```
String inTemp;
double userAnswer;
inTemp = stdin.readLineSync();
try {
  userAnswer = double.parse(inTemp);
} on FormatException {  // uh oh, could not be turned into double
  print("Could not interpret input.");
  return;
}
```

A literal number is determined as an int or double, based on the presence of a decimal point.

Strings

Strings are used for representing text. In Dart, strings can contain any character specified in the Unicode specification, which includes non-Latin characters, such as those found in Mandarin or Arabic. Strings are instances of the String class, which has several useful methods for manipulating them. Both double quotes (") and single quotes (') can be used for specifying String literals. Pick one or the other style and stick with it.

"hello" is equivalent to 'hello'.

Variable values can be interpolated within String literals, using $ for single variables and ${} for the result of an expression.

```
String weather = "It's $temperature degrees and cloudy.";
String countdown = "It's ${2069 - year} years from the big anniversary.";
```

The String method split() divides a String by some character and returns a List.

```
List fruits = "apple,cherry,orange".split(",");  // fruits is ["apple", "cherry", "orange"]
```

The trim() method removes whitespace on the edges of a String, and the toLowerCase() and toUpperCase() methods are often useful when comparing one String to another. The + operator can be used for concatenating one String to the end of another String.

```
String fullName = "John" + " " + "Smith";  // fullName will be "John Smith"
```

www.it-ebooks.info

Constants and Final Variables

The keyword const is used to specify that a variable's value will not be changed after its initial declaration/initialization. In other words, the value is wholly known at compile time, as opposed to runtime.

```
const WHEELS_ON_CAR = 4;
const List MY_FAVORITE_FLAVORS = const ["Apple", "Orange", "Grape"];
```

It is illegal to modify a constant after it has been declared. Constants can be optimized by the compiler. Final variables are immutable. They cannot be changed after they are initialized, although unlike constants, they can be initialized at runtime.

```
final int d = 5;  // d can now not be changed again
```

Giving Programs Structure

In Dart, functions are first-class citizens. Dart has several superb built-in data structures. Dart is fully object-based and has advanced object-oriented constructs. Dart provides ample facilities for giving programs structure.

Functions (Chapter 5)

Functions are defined with a title, parameters (if any), and a body. They can also specify their return type. Dart has special syntactic sugar for short, one-line functions.

```
add(var x, var y) {
  return x + y;  // control ends, value returned as return value
}
int add(int x, int y) {...}  // specifies return type and parameter types
void doSomething() {...}  // no return expected
int z = add(1, 2);  // add() called with arguments 1, 2 for parameters x, y
int luckyNumber() => 7;  // short syntax for single line functions, return is implied
```

Functions can appear within other functions. Anonymous functions are functions without a name.

```
void main() {
  String exclaimIt(String s) => s + "!";  // function within function
  print(exclaimIt("Hey"));
}
```

Optional parameters are parameters that do not need to be provided when a function is called. They are left up to the discretion of the caller. Dart supports both positional and named optional parameters. Optional parameters can have default values.

```
void repeat(String word, [int repetitions = 1, String exclamation = ""]) {
  for (int i = 0; i < repetitions; i++) {
    print(word + exclamation);  // the + operator can concatenate strings
  }
}  // could be called with repeat("Dog", 2, "!");
```

www.it-ebooks.info

```
void repeat(String word, {int repetitions: 1, String exclamation: ""}) {
  for (int i = 0; i < repetitions; i++) {
    print(word + exclamation);  // the + operator can concatenate strings
  }
}  // could be called with repeat("Dog", repetitions: 2, exclamation: "!");
```

Lists (Chapter 6)

List objects hold ordered, sequential data.

```
List l1 = [];  // a blank List
List l2 = new List();  // also a blank List
List m = [1, 3, 5];  // List created using literal
```

You can index into a `List` to grab a specific element by its index. `List` objects are zero-indexed.

```
int v = m[1];  // v is 3
```

The `add()` method adds an element to the end of a `List`, while `remove()` can be used to remove a specific element. Finally, `removeLast()` gets rid of the last element in a `List`. `List` objects can also be declared to contain a specific type with <> syntax.

```
List<String> sList = ["hello", "goodbye"];  // specifying that sList will contain Strings.
sList.add("heya");  // sList is now ["hello", "goodbye", "heya"]
sList.remove("goodbye");
sList.removeLast();  // sList is ["hello"]
```

Maps (Chapter 6)

Map objects associate data (values) with identifiers (keys).

```
Map a = {};  // empty Map
Map c = new Map();  // also an empty Map
Map nameAge = {"Matt": 27, "John": 18, "Sarah": 17, "Larry": 80};
Map employees = {345: {"name": "Donald Smith", "Department": "Accounting", "Salary": 1000},
                 220: {"name": "Mark Anderson", "Department": "Sales", "Salary": 950},
                 572: {"name": "Elizabeth Brahmen", "Department": "Marketing", "Salary": 975}};
```

Values can be looked up or set via their keys, with similar syntax to indexing into a `List`.

```
Map productPrice = {"Gum": 0.95, "Soda": 1.05, "Chips": 1.99};
double gumPrice = productPrice["Gum"];  //gumPrice is now 0.95
productPrice["Cookie"] = 0.50;  //a new key/value pair added to productPrice
productPrice["Soda"] = gumPrice;  //the value for the key "Soda" is now 0.95
```

As with `List` objects, `Map` objects can have their contained types specified at declaration.

```
Map<String, double> productPrice = {"Gum": 0.95, "Soda": 1.05, "Chips": 1.99};
```

www.it-ebooks.info

Sets (Chapter 6)

Set objects are for storing unique, unordered data. A set cannot contain two identical elements.

```
Set blankSet = new Set();  // empty set
```

As with List objects, Set objects can have their contained types specified at instantiation time.

```
Set<String> jerryColors = new Set.from(["blue", "red", "green"]);
```

Set objects are a convenient way to check for duplicates in a List.

```
bool containsDuplicates(List l) {
  Set s = new Set.from(l);  //create a new Set by converting the List
  if (s.length < l.length) {
    return true;
  }
  return false;
}
```

Defining Classes (Chapter 10, Chapter 11)

Classes are the fundamental building blocks of objects. They describe the methods, instance variables, and properties that a class contains. Class definitions additionally specify the relationship between one class and another. The constructor of a class is a special method that is used for initializing it (see Chapter 10 for some nuance on constructors). A variable or method specified with an underscore (_) is "private" to the library in which the class is defined.

```
class Dice {
  /// Instance Variables
  int _sides = 6;
  int _numberOfDice = 2;
  List<int> _values = [];

  /// Properties
  int get maximumValue => sides * numberOfDice;
  int get numberOfDice => _numberOfDice;
  int get sides => _sides;
  /// total is the sum of [_values]
  int get total => _values.fold(0, (first, second) => first + second);

  /// Constructor
  /// constructs a new Dice object, setting _sides and _numberOfDice
  Dice(this._sides, this._numberOfDice);

  /// Methods
  /// generate random values for [_values]
  void roll() {
    List newValues = [];
    Random rand = new Random();
```

www.it-ebooks.info

```
    for (int i = 0; i < numberOfDice; i++) {
      newValues.add(rand.nextInt(sides) + 1);  // number from 1 to sides
    }
    _values = newValues;
  }

  /// print the values of the dice
  void printDice() => print(_values);
}
```

The `abstract` keyword is used for defining an abstract class (a class that can't be instantiated and only serves as a superclass of other classes). The `extends` keyword is used for defining a class as the subclass of another class. Only one class can appear after `extends`.

```
abstract class Shape {
  double get perimeter;
  double get area;
  String get description;
}

class Circle extends Shape {
  double radius;
  Circle(this.radius);

  double get perimeter => radius * 2 * PI;
  double get area => PI * (radius * radius);
  String get description => "I am a circle with radius $radius";
}
```

Subclasses can refer to their superclass with the keyword `super.` `this` refers to the instance of the class currently being worked from within. Classes implicitly define their own interfaces. One class can implement another class's interface, by using the `implements` keyword. A class can implement multiple interfaces.

```
class A {
  void silly() {
    print("A's Silly");
  }
}

class B {
  void awesome() {
    print("B's Awesome");
  }
}

class C implements A, B {
  void silly() {
    print("C's Silly");
  }
```

www.it-ebooks.info

```
  void awesome() {
    print("C's Awesome");
  }
}
```

A class can be cast from one class to another, using the as operator.

```
A c = new C();
(c as C).awesome();
```

Classes that have no constructors, are subclasses of `Object`, and make no calls to `super` can be used as mixins. Mixins are functionality that is appended to another class. The `with` keyword is used to declare that a class uses a mixin.

```
class TimeStamp {
  DateTime creationTime = new DateTime.now();
  void printTimeStamp() {
    print(creationTime);
  }
}

class NewBorn extends Patient with TimeStamp {
  NewBorn(String name) : super(name);
}
```

`NewBorn` now has the `printTimeStamp()` method.

Libraries (Chapter 10)

Libraries are useful for packaging reusable Dart code. Defining a library compiles all of the files of the library into one package. All of the `import` statements for the files included in the library are specified in the library declaration. Each file includes `part of library_name` at its top.

```
library pig;

import "dart:math";  // for Random
import "dart:io";  // for stdin

part "dice.dart";
part "player.dart";
```

In `dice.dart`, you'll see:

```
part of pig;
```

Key Packages in the Standard Library

Dart comes with an extensive standard library, divided into logical bundled packages.

www.it-ebooks.info

dart:html (Chapter 8)

dart:html is the library used for manipulating the Document Object Module (DOM). Element is a key class. It is the base class for all of the classes in dart:html that are used to represent elements of an HTML document. querySelector() is a function used for grabbing DOM elements by their CSS class, id attribute, or tag type, based on their CSS selector. For example, a <div> tag with id fish could be grabbed with querySelector("#fish").

There are Element subclasses, such as ImageElement, CanvasElement, ButtonElement, etc., that represent specific HTML tags. Each of these classes has properties that correspond to the specific attributes of the HTML tag in question. In addition, all Element objects also have the methods getAttribute() and setAttribute() to manipulate them.

```
ImageElement ie = (querySelector("#fish_image") as ImageElement);
ie.src = "pike.png";
```

All Element objects have the generic properties text and innerHtml, which represent the string for display contained in an Element and the HTML that represents the tags within it, respectively.

```
DivElement myDiv = new DivElement();
ButtonElement myButton = new ButtonElement();
myDiv.innerHtml = "<strong>My Strong Text</strong>";
myButton.text = "Click Me!";
```

One Element gets added as a child of another, using the append() method.

```
myDiv.append(myButton);
```

Events that occur within the browser to an Element can be captured by adding listeners to various named ElementStream properties of an Element, such as onClick and onKeyDown, that process the events. Listeners are functions that have an object representing the event in question as a parameter.

```
myButton.onClick.listen((MouseEvent me) => ...);
```

Changes to the DOM that are made from a Dart program are implemented in real time by the browser. dart:html also includes extensive support for working with the HTML Canvas element. CanvasElement has a property, context2D, that has numerous methods for drawing graphical primitives to the screen.

```
CanvasRenderingContext2D myCanvasContext = myCanvas.context2D;
myCanvasContext.setFillColorRgb(255,0,0);  // RGB is Red, Green, Blue levels from 0-255 each
myCanvasContext.fillRect(myCanvas.width/2, myCanvas.height/2, 100, 200);  // x, y width, height
```

Quick error messages can be displayed to the screen as dialog boxes, with window.alert().

```
window.alert("Error: invalid input.");
```

Table A-5 shows some common HTML tags/elements and their respective classes in dart:html.

www.it-ebooks.info

Table A-5. *Common HTML Tags and Their `dart:html` Element Subclasses*

Tag	Description	dart:html Element
`<a>`	Anchor (link)	`AnchorElement`
`<div>`	Div	`DivElement`
`<span>`	Span	`SpanElement`
`<p>`	Paragraph	`ParagraphElement`
`<img>`	Image	`ImageElement`
`<canvas>`	HTML Canvas	`CanvasElement`
`<input>`	Input form	`InputElement`
`<ul>`	Unordered List	`UListElement`
`<ol>`	Ordered List	`OListElement`
`<li>`	List Item	`LIElement`
`<button>`	Button	`ButtonElement`

dart:io (Chapter 4, Chapter 14)

Reading input from the command line is easy using `stdin.readLineSync()`.

```
String temp = stdin.readLineSync();  // read in from the keyboard
```

Remember to check that the input is really formatted as you want it, before using it. The `File` class can be used for reading files in a command-line application.

```
File file = new File("pandp.txt");
file.readAsString().then((String fileContent) => print(fileContent));
```

dart:math

The `Random` class is useful for generating random numbers, using its methods `nextInt()` and `nextDouble()`.

```
Random rand = new Random();
int choice = rand.nextInt(3);         // creates random integer between 0 and 2
double choice2 = rand.nextDouble();  // between 0.0 (inclusive) and 1.0 (exclusive)
```

The built-in constant `PI` represents the special number pi.

```
print(PI);
```

The class `Point` represents a point on a two-dimensional plane. Every `Point` object has an x and a y coordinate expressed as properties. The `Point` class has its +, -, * and == operators overloaded.

```
Point a = new Point(2, 2);
Point b = new Point(4, 4);
b = b - a;
print(b);  // prints Point(2, 2)
```

www.it-ebooks.info

```
a = a * 2;
print(a);  // prints Point(4, 4)
a = b + a;
print(a);  // prints Point(6, 6);
if ((b + b) == (a - b)) {  // this is true
  print("Both sides of the equation hold equivalent values.");
}
```

Rectangle is a class for representing geometric rectangles. A Rectangle is initialized, based on its top-left corner, width, and height. It has the very useful methods containsPoint() and intersects() to check its relationship with points and other rectangles in the two-dimensional plane.

For comparing numbers, dart:math provides the convenience functions max() and min(), which operate on any two num objects.

unittest (Chapter 13)

unittest does not come prepackaged with the Dart SDK. You need to add it to your pubspec.yaml before using it and import it as import 'package:unittest/unittest.dart';.

The test() function is the core of unittest. It takes a String name for a test and a function defining a test. Calls of expect() within a test define what it means to pass a test.

```
test("exclaim() test", (){
  String original = "I'm testing";
  expect(exclaim(original), equals("I'm testing!"));
});
```

expect() has several matchers like equals(), which is seen above. The first parameter of expect() is the object being tested and the second is the matcher. Table A-6 shows several common matchers.

Table A-6. *Selected Common Matchers in the unittest Library*

Matcher	Parameter Type
isTrue	None
isFalse	None
isNull	None
isNotNull	None
isEmpty	None
equals()	Object
greaterThan()	num
lessThan()	num
closeTo()	num, num (the latter is the delta allowed)
equalsIgnoringCase()	String

(*continued*)

www.it-ebooks.info

Table A-6. (*continued*)

Matcher	Parameter Type
`equalsIgnoringWhitespace()`	`String`
`matches()`	`RegExp`
`orderedEquals()`	`Iterable`
`unorderedEquals()`	`Iterable`
`containsValue()` (used with a `Map`)	`Object`

Multiple tests can be put together within a single group. Groups are defined using calls of the `group()` function.

```
group("exclaim tests", () {
  List testCases = [["Dog", "Dog!"], ["", "!"], ["H e l l o ", "H e l l o !"]];
  for (List testCase in testCases) {
    test(testCase[0], (){
      expect(exclaim(testCase[0]), equals(testCase[1]));
    });
  }
});
```

`group()` takes the name of the group as its first parameter and a function where tests are defined as its second parameter. Groups are helpful for organization purposes. When tests are run, the tests within the same group will be nicely outputted together.

dart:async (Chapter 8, Chapter 14)

The `Timer` class of `dart:async` is useful for scheduling things that need to occur periodically, or things that need to occur just one time in the future.

```
Timer t = new Timer(const Duration(seconds: 2), () {
  clickedCard.src = CARD_BACK;
  tempClicked.src = CARD_BACK;
});  // one time
```

This `Timer` attempts to execute `update()` every 17 milliseconds. The actual accuracy of that time period depends on whether other code running is easily interrupted.

```
Timer t = new Timer.periodic(const Duration(milliseconds:17), update);
```

`dart:async` is also the home of `Future`, a key class used throughout the Dart standard library. It is used for getting the result of a computation after an asynchronous task has completed. The `then()` method of `Future` is called with a function that will execute, once the `Future` completes, including its result as a parameter.

```
Future<String> f = HttpRequest.getString("pandp.txt");
f.then((String s) => print(s));
```

`Future` objects have built-in mechanisms for catching errors.

```
f.then((String s) => print(s)).catchError((Error e) => print(e.toString()));
```

www.it-ebooks.info

Finally, Future objects can be strung together.

```
File file = new File("pandp.txt");
file.readAsString().then((String fileContent) {
  print(fileContent);
  return new File("pandp.txt").readAsString();
}).then((String fileContent) => print(fileContent));
```

dart:isolate (Chapter 14)

An isolate is an independent, concurrent process, with its own separate memory space. Isolates are spawned at the point of a function call. They use ReceivePort and SendPort objects for intra-process communication. The SendPort method send() is used for sending data across the wire. The ReceivePort method listen() asynchronously receives data coming across the wire.

```
void calcPi(SendPort sp) {
  ...
  sp.send(pi);  // send the result back
}

void main() {
  ReceivePort rp = new ReceivePort();
  rp.listen((data) {  // data is what we receive from sp.send()
    print("Pi is $data");
    rp.close();  // we're done, close up shop
  });
  Isolate.spawn(calcPi, rp.sendPort);  // start the Isolate
}
```

General Style Conventions

While style is not enforced, it's good practice to follow style conventions when they make sense. This book broke from general Dart style recommendations by declaring local variables with their type, instead of with the generic var. This was done to ease the beginner's type understanding and, subjectively, to improve code clarity. However, there were many style conventions that were rigorously followed—and you should follow them too.

- Class names should be written in uppercase CamelCase.
- Class instance variables, properties, and method names should be written with lowercase camelCase.
- Constants should be written in ALL_UPPERCASE_WITH_UNDERSCORES_FOR_SPACES.
- Variable names should be descriptive.
- Code should be well-commented.
- There should be spacing around operators.
- Indentations should be two spaces.
- When in doubt, use Dart Editor's "Format" command, available in the contextual menu, brought up by right-clicking in an editor window.

For more Dart style recommendations, check out the article "Dart Style Guide" by Bob Nystrom at www.dartlang.org/articles/style-guide.

www.it-ebooks.info