Chapter II

All About Dart Language

I. Resources for Dart (Read First)

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

https://dartpad.dev/

II. Important Concepts

As you learn about the Dart language, keep these facts and concepts in mind:

- Everything you can place in a variable is an object, and every object is an instance of a class. Even numbers, functions, and null are objects. All objects inherit from the Object class.

- Although Dart is strongly typed, type annotations are optional because Dart can infer types. In the code above, number is inferred to be of type int. When you want to explicitly say that no type is expected, use the special type dynamic.

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

- Dart supports top-level functions (such as main()), as well as functions tied to a class or object (static and instance methods, respectively). You can also create functions within functions (nested or local functions).

- Similarly, Dart supports top-level variables, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as fields or properties.

- Unlike Java, Dart doesn't have the keywords public, protected, and private. If an identifier starts with an underscore (_), it's private to its library.

- Identifiers can start with a letter or underscore (_), followed by any combination of those characters plus digits.

- Dart has both expressions (which have runtime values) and statements (which don't). For example, the conditional expression condition ? expr1 : expr2 has a value of expr1 or expr2. Compare that to an if-else statement, which has no value. A statement often contains one or more expressions, but an expression can't directly contain a statement.

- Dart tools can report two kinds of problems: warnings and errors. Warnings are just indications that your code might not work, but they don't prevent your program from executing. Errors can be either compile-time or run-time. A compile-time error prevents the code from executing at all; a run-time error results in an exception being raised while the code executes.

III.Keywords & Identifiers

Keywords

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

abstract 2	dynamic 2	implements 2	show 1
as 2	else	import 2	static 2
assert	enum	in	super
async 1	export 2	interface 2	switch
await 3	extends	is	sync 1
break	external 2	library 2	this
case	factory 2	mixin 2	throw
catch	false	new	true
class	final	null	try
const	finally	on 1	typedef 2
continue	for	operator 2	var
covariant 2	Function 2	part 2	void
default	get 2	rethrow	while
deferred 2	hide 1	return	with
do	if	set 2	yield 3

Avoid using these words as identifiers. However, if necessary, the keywords marked with superscripts can be identifiers:

- Words with the superscript 1 are contextual keywords, which have meaning only in specific places. They're valid identifiers everywhere.

- Words with the superscript 2 are built-in identifiers. To simplify the task of porting JavaScript code to Dart, these keywords are valid identifiers in most places, but they can't be used as class or type names, or as import prefixes.

- Words with the superscript 3 are newer, limited reserved words related to the asynchrony support that was added after Dart's 1.0 release. You can't use await or yield as an identifier in any function body marked with async, async*, or sync*.

All other words in the table are **reserved words**, which can't be identifiers.

Identifiers

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

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

Eg.

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

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

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

Note:

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

IV.Default Value

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

Eg.

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

V. Final & Constant Value

If you never intend to change a variable, use `final` or `const`, either instead of `var` or in addition to a type. A final variable can be set only once; a const variable is a compile-time constant. (Const variables are implicitly final.) A final top-level or class variable is initialized the first time it's used.

Eg.

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

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

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

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

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

Use `const` for variables that you want to be **compile-time constants**. If the const variable is at the class level, mark it `static const`.

VI.Operator

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

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

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

VII.Identifier In Detail

int

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

double

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

Eg.

```
var x = 1;

var hex = 0xDEADBEEF;

var y = 1.1;

var exponents = 1.42e5;

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

```
//Type Conversion

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


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


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

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

const msPerSecond = 1000;

const secondsUntilRetry = 5;

const msUntilRetry = secondsUntilRetry * msPerSecond;
```

Strings

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

Eg.

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.”;
```

You can put the value of an expression inside a string by using ${expression}. If the expression is an identifier, you can skip the {}. To get the string corresponding to an object, Dart calls the object's toString() method.

Eg.

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

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


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

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

//String Multi Line

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

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


//Raw String  : Start with small r

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


print(str);

print(rawStr);
```

Booleans (the boolean literals true and false are used)

Lists

Perhaps the most common collection in nearly every programming language is the array, or ordered group of objects. In Dart, arrays are List objects and use zero-based indexing.eg

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

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

//The spread operator (...)

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

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

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

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

//if in list , conditionally

```
var nav = [
  'Home',
  'Furniture',
  'Plants',
  if (promoActive) 'Outlet'
];
```

Here's an example of using collection for to manipulate the items of a list before adding them to another list:

//For in the list iterated

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

Sets

A set in Dart is an unordered collection of unique items. Dart support for sets is provided by set literals and the Set type.

Eg.

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

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

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

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

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

Note:

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

Maps

In general, a map is an object that associates keys and values. Both keys and values can be any type of object. Each key occurs only once, but you can use the same value multiple times.

Eg.

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

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

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

//Creating the same objects using a Map constructor:

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

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

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


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

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

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

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

Runes

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

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world's writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing 32-bit Unicode values within a string requires special syntax.

The usual way to express a Unicode code point is \uXXXX, where XXXX is a 4-digit hexadecimal value. For example, the heart character (♥) is \u2665.

To specify more or less than 4 hex digits, place the value in curly brackets. For example, the laughing emoji (😆) is \u{1f600}.

Symbols

A Symbol object represents an operator or identifier declared in a Dart program. You might never need to use symbols, but they're invaluable for APIs that refer to identifiers by name, because minification changes identifier names but not identifier symbols.

To get the symbol for an identifier, use a symbol literal, which is just # followed by the identifier:

```
#radix
#bar
```

Symbol literals are compile-time constants.

Eg.

```
var mysymbol = 1;

print(#mysymbol);
```