Department of Information Engineering Technology
The Superior University, Lahore

Mobile Application Development Lab

Experiment No.2

Implement Dart Basics

Prepared for

EISHA NAWAZ

By:

Name: _____

ID: _____

Section: _____

Semester: _____

Total Marks: _____

Obtained Marks: _____

Signature: _____

Date: _____

Experiment No.2
Implement Dart Basics
(Rubrics)

Name: _____ **Roll No:** _____

A. PSYCHOMOTOR

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Follow Procedures	1	0	0.25	0.5	0.75	1	
2	Software and Simulations	2	0	0.5	1	1.5	2	
3	Accuracy	3	0	0.75	1.5	2.25	3	
Sub Total		6	Sub Total marks Obtained in Psychomotor(P)					

B. AFFECTIVE

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Respond	1	0	0.25	0.5	0.75	1	
2	Lab Report	1	0	0.25	0.5	0.75	1	
3	Assigned Task	2	0	0.5	1	1.5	2	
Sub Total		4	Sub Total marks Obtained in Affective (A)					

Instructor Name: EISHA NAWAZ **Total Marks (P+A):** _____

Instructor Signature: _____ **Obtained Marks (P+A):** _____

Lab 2 Contents

Implement Dart Basics

Objective:

- Execute core Dart programming concepts, including variables, collections (Lists/Maps), and functions, with technical precision.
- Implement a structured Flutter Widget Tree by accurately nesting layout and content widgets to build a functional user interface.
- Translate terminal-based Dart logic into a visual mobile application, demonstrating a professional grasp of modern development frameworks.

Dart is the foundation of learning to develop Flutter projects. you'll create apps that implement these concepts.

WHY USE DART?

Before you can start developing Flutter apps, you need to understand the programming lan- guage used, namely, Dart. Google created Dart and uses it internally with some of its big products such as Google AdWords. Made available publicly in 2011, Dart is used to build mobile, web, and server applications. Dart is productive, fast, portable, approachable, and most of all reactive.

Dart is an object-oriented programming (OOP) language, has a class-based style, and uses a C-style syntax. If you are familiar with the C#, C++, Swift, Kotlin, and Java/JavaScript languages, you will be able to start developing in Dart quickly. But don't worry—even if you are not familiar with these other languages, Dart is a straightforward language to learn, and you can get started relatively quickly.

What are some of the benefits of using Dart?

- Dart is *ahead-of-time* (AOT) compiled to native code, making your Flutter app fast. In other words, there's no intermediary to interpret one language to another, and there are no bridges. AOT compilation is used when compiling your app for release mode (such as to the Apple App Store and Google Play).
- Dart also is *just-in-time* (JIT) compiled, making it fast to display your code changes such as via Flutter's stateful hot reload feature. JIT compilation is used when debugging your app by running it in the

simulator/emulator.

- Since Flutter uses Dart, all the sample user interface (UI) code in this book is written in Dart, removing the need to use separate languages (markup, visual designer) to cre- ate the UI.
- Flutter rendering runs at 60 frames per second (fps) and 120fps (for capable devices of 120Hz). The more fps, the smoother the app.

COMMENTING CODE

In any app, comments help the readability of the code, as long as they're not overdone. Comments can be used to describe the logic and dependencies of the app.

There are three types of comments: single-line, multiline, and documentation comments. Single-line comments are commonly used to add a short description. Multiline comments are best suited for long descriptions that span multiple lines. Documentation comments are used to fully document a piece of code logic, usually giving detailed explanations and sample code in the comments.

Single-line comments begin with //, and the Dart compiler ignores everything to the end of the line.

```
// Retrieve from the database the list filtered by company
_listOrders.get(...
```

Multiline comments begin with /* and end with */. The Dart compiler ignores everything between the slashes.

```
/*
 * Allow users to filter by multiple options
_listOrders.get(filterBy: _userFilter...
*/
```

Documentation comments begin with ///, and the Dart compiler ignores everything to the end of the line unless enclosed in brackets. Using brackets, you can refer to classes, methods, fields, top-level variables, functions, and parameters. In the following example, the generated documentation, [FilterBy], becomes a link to the API documentation for the class. You can use the SDK's documen- tation generation tool (dartdoc) to parse Dart code and generate HTML

documentation.

```
/// Multiple filter options
///
/// Different
[FilterBy]
enum FilterBy
{COMPANY,
CITY, STATE
}
```

RUNNING THE MAIN() ENTRY POINT

Every app must have a top-level main() function, which is the entry point to the app. The main() function is where the app execution starts and returns a void with an optional List<String> param- eter for arguments. Each function can return a value, and for the main() function the data return type is a void (empty, contains nothing), meaning that it does not return a value.

In the following code, you see three different ways to use the main() function, but in all the exam- ple projects in this book, you will be using the first example—the arrow syntax void main() => runApp(MyApp());. All three ways to call the main() function are acceptable, but I prefer using the arrow syntax since it keeps the code on one line for better readability. However, the main reason to use the arrow syntax is that in all the example projects there is no need to call multiple statements. The arrow syntax => runApp(MyApp()) is the same as { runApp(MyApp()); }.

```
// arrow syntax
void main() => runApp(MyApp());

// or
void
    main()
    {
    runApp(
    MyApp
    ());
}

// or with a List of Strings
parameters void
```

```
main(List<Strings> filters) {
    print('filters: $filters');
}
```

REFERENCING VARIABLES

In the previous section, you learned that main() is the top-level entry to an app, and before you start writing code, it's important to learn about Dart variables. Variables store *references* to a value. Some of the built-in variable types are numbers, strings, Booleans, lists, maps, and runes. You can use var to declare (you will learn declaring variables in the next section) a variable without specifying the type. Dart infers the type of variable automatically. Although there is nothing wrong with using var, as a personal preference, I usually stay away from using it unless I need to do so. Declaring the variable type makes for better code readability, and it's easier to know which type of value is expected. Instead of using var, use the variable type expected: double, String, and so on. (The vari- able types are covered in the "Declaring Variables" section.)

An uninitialized variable has a value of null. When declaring a variable without giving it an initial value, it's called *uninitialized*. For example, a variable of type String is declared like String book- Title; and is uninitialized because the bookTitle value equals null (no value). However, if you declare it with an initial value of String bookTitle = 'Beginning Flutter', the bookTitle value equals 'Beginning Flutter'.

Use final or const when the variable is not intended to change the initial value. Use const for vari- ables that need to be *compile-time constants*, meaning the value is known at compile time.

DECLARING VARIABLES

Now you know that variables store references to a value. Next, you'll learn different options for declaring variables.

In Dart, all variables are declared public (available to all) by default, but by starting the variable name with an underscore (_), you can declare it as private. By declaring a variable private, you are saying it cannot be accessed from outside classes/functions; in other words, it can be used only from within the declaration class/function. (You will learn about classes and functions in the "Functions" and "Classes" sections later in this chapter.)

Note some built-in Dart variable types are lowercase like double and some uppercase like String.

What if the value of a variable doesn't need to change? Begin the declaration of the variable with final or const. Use final when the value is assigned at runtime (can be changed by the user). Use const when the value is known at compile time (in code) and will not change at runtime.

```
// Declared without specifying the type - Infers
type var filter = 'company';

// Declared by type
String filter = 'company';

// Uninitialized variable has an initial value of null
String filter;

// Value will not
change final
filter =
'company';

// or
final String filter = 'company';

// or
const String filter = 'company';

// or
const String filter = 'company' + filterOption;

    // Public variable (variable name starts without underscore)
    String userName = 'Sandy';

    // Private variable (variable name starts with
    underscore) String _userID = 'XW904';
```

Numbers

Declaring variables as numbers restricts the values to numbers only. Dart allows numbers to be int (integer) or double. Use the int declaration if your numbers do not require decimal point preci- sion, like 10 or 40. Use the

doubledeclaration if your numbers require decimal point precision, like 50.25 or 135.7521. Both int and double allow for positive and negative numbers, and you can enter extremely large numbers and decimal precision since they both use 64-bit (computer memory) values.

```
// Integer
int
counter =
0; double
price =
0.0; price
= 125.00;
```

Strings

Declaring variables as String allows values to be entered as a sequence of text characters. To add a single line of characters, you can use single or double quotes like 'car' or "car". To add multiline characters, use triple quotes, like "'car"'. Strings can be concatenated (combined) by using the plus (+) operator or by using adjacent single or double quotes.

```
// Strings
String defaultMenu = 'main';

// String concatenation
String combinedName = 'main' + ' ' +
'function'; String combinedNameNoPlusSign =
'main' ' 'function';

// String multi-line
String multilineAddress = '"
  123
  Any
  Street
  City,
  State,
  Zip
"';
```

Booleans

Declaring variables as bool(Boolean) allows a value of trueor falseto be entered.

```
// Booleans
```

```
bool isDone =
false; isDone =
true;
```

Lists

Declaring variables as List (comparable to arrays) allows multiple values to be entered; a List is an ordered group of objects. In programming, an array is an iterable (accessed sequentially) collection of objects, with each element accessible by the index position or a key. To access elements, the List uses zero-based indexing, where the first element index is at 0, and the last element is at the Listlength (number of rows) minus 1 (since the first index is 0, not 1).

A List can be of fixed length or growable, depending on your needs. By default, a Listis created as growable by using List()or []. To create a fixed-length List, you add the number of rows required by using this format: List(25). The following example uses string interpolation for the print statement: print('filter: $filter'). The $ sign before the variable converts the expression value to a string.

```
// List Growable
List contacts = List();

// or
List contacts = [];
List contacts = ['Linda', 'John', 'Mary'];

// List fixed-
length List
contact =
List(25);

// Lists - In Dart List is an array
List listOfFilters = ['company', 'city', 'state'];
listOfFilters.forEach((filter) {
    print('filter: $filter');
});
// Result from print statement
// filter: company
// filter: city
```

```
// filter: state
```

Maps

Maps are invaluable in associating a List of values by a Key and a Value. Mapping allows recalling values by their Key ID. The Key and Value can be any type of object, such as String, Number, and so on. Keep in mind that the Key needs to be unique since the Value is retrieved by the Key.

```
// Maps - An object that associates keys and values.
// Key: Value - 'KeyValue': 'Value'
Map mapOfFilters = {'id1': 'company', 'id2': 'city', 'id3': 'state'};

// Change the value of third item with Key of id3
mapOfFilters['id3'] = 'my filter';

print('Get filter with id3: ${mapOfFilters['id3']}');
// Result from print statement
// Get filter with id3: my filter
```

Runes

In Dart, declaring variables as Runes are the UTF-32 code points of a String. Emojis, anyone?

Unicode defines a numeric value for each letter, digit, and symbol. Dart uses the sequence of UTF-16 code units to represent a 32-bit Unicode value from a string require a special syntax (\uXXXX).
A Unicode code point is \uXXXX, where XXXX is a four-digit hexadecimal value. Runes return the integer value of the Unicode; then you use String.fromCharCodes() to allocate a new String for the specified charCode.

```
// Emoji smiling angel Unicode is u+1f607
// Remove the Plus sign and replace with curly
brackets Runes myEmoji =
Runes('\u{1f607}');
print(myEmoji);
// Result from print statement
// (128519)

print(String.fromCharCodes(myEmoji));
// Result from print statement
// 😇
```

USING OPERATORS

An operator is a symbol used to perform arithmetic, equality, relational, type test, assignment, logical, conditional, and cascade notation. Tables 3.1 through 3.7 go over some of the common operators.
For the sample code, I use the values directly to simplify the examples instead of using variables.

TABLE 3.1: Arithmetic operators

OPERATOR	DESCRIPTION	SAMPLE CODE
+	Add	7 + 3 = 10
-	Subtract	7 - 3 = 4
*	Multiply	7 * 3 = 21
/	Divide	7 / 3 = 2.33

TABLE 3.2: Equality and relational operators

OPERATOR	DESCRIPTION	SAMPLE CODE
==	Equal	7 == 3 = false
!=	Not equal	7 != 3 = true
>	Greater than	7 > 3 = true
<	Less than	7 < 3 = false
>=	Greater than or equal to	7 >= 3 = true 4 >= 4 true
<=	Less than or equal to	7<= 3 = false 4 <= 4 = true

TABLE 3.3: Type test operators

OPERATOR	DESCRIPTION	SAMPLE CODE
as	Typecast like import library prefixes.	import 'travelpoints .dart' as travel;
is	If the object contains the specified type, it evaluates to true.	if (points is Places) = true
is!	If the object contains the specified type, it evaluates to false(not usually used).	if (points is! Places) = false

TABLE 3.4: Assignment operators

OPERATOR	DESCRIPTION	SAMPLE CODE
=	Assigns value	7 = 3 = 3
??=	Assigns value only if variable being assigned to has a value of null	Null ??= 3 = 3 7 ??= 3 = 7
+=	Adds to current value	7 += 3 = 10
-=	Subtracts from current value	7 -= 3 = 4
*=	Multiplies from current value	7 *= 3 = 21
/=	Divides from current value	7 /= 3 = 2.33

TABLE 3.5: Logical operators

OPERATOR	DESCRIPTION	SAMPLE CODE
!	!is a logical 'not'. Returns the opposite value	if (!(7 > 3)) = false

	of the variable/expression.	
&&	&&is a logical 'and'. Returns trueif the values of the variable/expression are all true.	if ((7 > 3) && (3 < 7)) = true if ((7 > 3) && (3 > 7)) = false
!!	!!is a logical 'or'. Returns trueif at least one value of the variable/expression is true.	if ((7 > 3) || (3 > 7)) = true if ((7 < 3) || (3 > 7)) = false

TABLE 3.6: Conditional expressions

OPERATOR	DESCRIPTION	SAMPLE CODE
condition ? value1 : value2	If the condition evaluates to true, it returns value1. If the condition evaluates to false, it returns value2. The value can also be obtained by calling methods.	(7 > 3) ? true : false = true (7 < 3) ? true : false = false

TABLE 3.7: Cascade notation (..)

OPERATOR	DESCRIPTION	SAMPLE CODE
..	The cascade notation is represented by double dots (..) and allows you to make a sequence of operations on the same object.	Matrix4.identity() ..scale(1.0, 1.0) ..translate(30, 30);

USING FLOW STATEMENTS

To control the logic flow of the Dart code, take a look at the following flow statements:

- ifand elseare the most common flow statements; they decide which code to run by com- paring multiple scenarios.

- The ternary operator is similar to the if and else statements but used when only two choices are needed.
- for loops allow iterating a List of values.
- while and do-while are a common pair. Use the while loop to evaluate the condition before running the loop, and use do-while to evaluate the condition after the loop.
- while and break are useful if you need to stop evaluating the condition in the loop.
- continue is for when you need to stop the current loop and start the next loop iteration.
- switch and case are alternatives to the if and else statements, but they require a default clause.

if and else

The if statement compares an expression, and if true, it executes the code logic. The expression is wrapped by open and close parentheses followed by the code logic wrapped in braces. The if state- ment also supports multiple optional else statements, which are used to evaluate multiple scenarios. There are two types of else statements: else if and else. You can use multiple else if state- ments, but you can have only one else statement, usually used as a catchall scenario.

In the following example, the if statement is checking whether the store is open or closed and whether items are out of stock or nothing matched. isClosed, isOpen, and isOutOfStock are bool variables. The first if statement checks whether the isClosed variable is true, and if yes, it prints to the log 'Store is closed'. How does it know you are checking for true or false without the equality operator? When checking for bool values, the if statement checks by default if the variable is true; this is the equivalent of isClosed == true. To check whether a variable is false, you can use the not equal (!=) operator like isClosed != true or the equality (==) operator like isClosed == false. The else if statement (isOpen) checks whether the isOpen variable equals true, and it's the same for the else if (isOutOfStock) variable. The last

else statement does not have a condi- tion; it's a catchall scenario if none of the other conditions is met.

```
// If
and
else if
(isClos
ed) {
    print('Store is closed');
}
else if (isOpen) {
    print('Store is
    open');
}
else if (isOutOfStock) {
    print('Item is out of
    stock');
}
else {
    print('Nothing matched');;
}
```

ternary operator

The ternary operator takes three arguments, and it's usually used when only two actions are needed. The ternary operator checks the first argument for comparison, the second is the action if the argu- ment is true, and the third is the action if the argument is false(see Table 3.8).

TABLE 3.8: ternary operator

COMPARISON		TRUE		FALSE
isClosed	?	askToOpen()	:	askToClose()

This will look familiar to you from the "Operators" section's conditional expressions because it's used often to make code flow decisions.

```
// Shorter way of if and else
statement isClosed ? askToOpen() :
askToClose();
```

for Loops

The standard for loop allows you to iterate a List of values. Values are obtained by restricting the number of loops by a defined length. An example is to loop through the top three values, which means you specify the number of times to execute the loop. Using a List of values also allows you to use the for-in type of Iteration. The Iteration class needs to be of type Iterable (a collection of values), and the List class conforms to this type. Unlike the standard for loop, the for-in loop iterates through every object in the List, exposing each object's properties values.

Let's take a look at two examples showing how to use the standard for loop and the for-in loops. In the first example, you'll use the standard for loop and iterate through a List of String values with the listOfFilters variable. The standard for loop takes three parameters.

- The first parameter initializes the variable i as an int variable counting each loop executed.

  Since the List uses zero-based indexing, the i variable is initialized with 0 and not 1.

- The second parameter controls how many times to loop through the List by comparing the current number of loops (i) to the total number of loops (listOfFilters.length) to exe- cute. Since the List uses zero-based indexing, the i variable value has to be less than the number of rows in the List.

- The third parameter increases the number of loops executed by increasing the i variable with each loop. Inside the loop, the print statement is used to show each value from the listOfFilters List.

```
// Standard for loop
List listOfFilters = ['company', 'city', 'state'];
for (int i = 0; i < listOfFilters.length; i++)
{
```

```
    print('listOfFilters: ${listOfFilters[i]}');
}
// Result from print statement
// listOfFilters: company
// listOfFilters: city
// listOfFilters: state
```

In the following example, you'll use the for-in loop and iterate through a List of int values with the listOfNumbers variable. The for-in loop takes one parameter that exposes the object (listOf- Numbers) properties. You declare the int number variable to access the properties of the listOfNum- bers List. Inside the loop, the print statement is used to show each value from listOfNumbers by using the number variable value.

```
// or for-in loop
List listOfNumbers = [10, 20, 30]; for (int number in listOfNumbers) {
    print('number: $number');
}
// Result from print statement
// number: 10
// number: 20
// number: 30
```

while and do-while

Both the while and do-while loops evaluate a condition and continue to loop as long as the condi- tion returns a value of true. The while loop evaluates the condition before the loop is executed.

The do-while loop evaluates the condition after the loop is executed at least once. Let's look at two examples that show how to use the while and do-while loops.

In both examples, the askToOpen() method is called in the loop, executing logic that sets the isClosed variable as a bool value of true or false. Use the while loop if you already have enough information to evaluate the condition (isClosed) before the loop is executed. Use the do-while
if you need to execute the loop first before you have enough information to evaluate the condition (isClosed).

In the first example, you'll use the while loop and iterate as long as the isClosed variable returns a value of true. In this case, the loop continues to execute as long as the isClosed variable is true and continues to loop. Once the isClosed variable returns false, the while stops from executing the next loop.

```
// While - evaluates the condition before the
loop while (isClosed){
    askToOpen();
}
```

In the second example, you'll use the do-while loop and iterate as long as the isClosed variable returns a value of true, like the first example. The loop is first executed at least once; then the condi- tion is evaluated, and as long as it returns true, it continues to loop. Once the isClosed variable returns false, do-while stops from executing the next loop.

```
// Do While - evaluates the condition after the
loop do {
askToOpen();
} while (isClosed);
```

while and break

Using the break statement allows you to stop looping by evaluating a condition inside the while loop.

In this example, the askToOpen() method is called inside the loop by the if statement, executing logic that returns a bool value of true or false. As long as the value returned is false, the loop continues as normal by calling the checkForNewOrder() method. But once askToOpen() returns a value of true, the break statement is executed, stopping the loop. The checkForNewOrder() method is not called, and the entire while statement stops from running again.

```
// Break - to
stop loop
while
(isClosed) {
    if (askToOpen()) break;
```

```
    checkForNewOrder();
}
```

continue

By using the continue statement, you can stop at the current loop location and skip to the start of the next loop iteration.

In this example, the for statement loops through a List of numbers from 10 to 80. Inside the loop, the if statement checks whether the number is less than 30 and greater than 50, and if the condition is met, the continue statement stops the current loop and starts the next iteration. Using the print statement, you see that only the numbers 30, 40, and 50 are printed to the log.

```
// Continue - skip to the next loop iteration
List listOfNumbers = [10, 20, 30, 40, 50, 60, 70,
80]; for (int number in listOfNumbers) {
    if (number < 30 || number >
        50) { continue;
    }
    print('number: $number'); // Will print number 30, 40, 50
}
```

switch and case

The switch statement compares integer, string, or compile-time constants using == (equality). The switch statement is an alternative to the if and else statements. The switch statement evaluates an expression and uses the case clause to match a condition and executes code inside the matching case. Each case clause ends by placing a break statement as the last line. It's not commonly used, but if you have an empty (no code) case clause, the break statement is not needed since Dart allows
it to fall through. If you need a catchall scenario, you can use the default clause to execute code that is not matched by any of the case clauses, placed after all the case clauses. The default clause does not require a break statement. Make sure that the last case is a default clause that executes logic if no previous case clause has a match.

In our example, we have the String coffee variable initialized to the 'espresso' value. The switch statement uses the coffee variable expression

where each case clause needs to match the coffee variable value. When the case clause matches the correct value, the code associated with the clause is executed. If none of the case clauses matches the coffee variable value, the defaultclause is selected and executes the associated code.

```
// switch and case
String coffee = 'espresso'; switch (coffee) {
    case 'flavored':
        orderFlavored();
        break;
    case 'dark-roast':
        orderDarkRoast();
        break;
    case 'espresso':
        orderEspresso();
        break;
    default:
        orderNotAvailable();
}
```

USING FUNCTIONS

Functions are used to group reusable logic. A function can optionally take parameters and return values. I love this feature. Because Dart is an object-oriented language, functions can be assigned to variables or passed as arguments to other functions. If the function executes a single expression,

you can use the arrow (=>) syntax. All functions return a value by default, and if no return statement
is specified, Dart automatically appends to the function body the return null statement, which is implicitly added for you.

Since all functions return a value, you start each function by specifying the return type expected. When calling a function and a return value is not needed, then start the function with the void type, meaning nothing. Using the void type is not required, but it's recommended for readability. But when the function is expected to return a value, start the function with the type of data being passed back (bool, int, String, List. . .) and use the return statement to pass a value.

The following examples show different ways to create/call functions and return different types of values. The first example shows that the app's main() is a function with void as the return type.

```
// Functions - Our main() is a
function void main() =>
runApp(new MyApp());
```

The second example has a void as the return type, but the function takes an int as a parameter, and when the code is executed, the print statement shows the value to the log terminal. Since the func- tion is expecting a parameter, you call it by passing the value like orderEspresso(3).

```
// Function - pass value
void orderEspresso(int
    howManyCups) { print('Cups #:
    $howManyCups');
}
orderEspresso(3);
// Result from print statement
// Cups #: 3
```

The third example builds upon the second example of receiving a parameter and returns a bool value as a return type. Just after the function, a bool isOrderDone variable is initialized by calling the function and passing a value of three; then the print statement shows the bool value sent back by the function.

```
// Function - pass value and return
value bool orderEspresso(int
howManyCups) {
    print('Cups #:
    $howManyCups'); return
    true;
}
bool isOrderDone =
orderEspresso(3); print('Order
Done: $isOrderDone');
// Result from print statement
// Cups #: 3
// Order Done: true
```

The fourth example builds upon the third example by making the function parameter optional by wrapping the [int howManyCups] variable inside square brackets.

```
// Function - pass optional value and return value
// Optional value is enclosed in square brackets
[] bool orderEspresso1([int howManyCups])
{
    print('Cups #:
    $howManyCups'); bool
    ordered = false;
        if (howManyCups !=
            null) { ordered =
            true;
        }
    return ordered;
        }
        bool isOrderDone1 =
        orderEspresso1(); print('Order
        Done1: $isOrderDone1');
        // Result from print statement
        // Cups #: null
        // Order Done: false
```

IMPORT PACKAGES

To use an external package, library or an external class, use the import statement. Separating code logic into different class files allows you to separate and group code into manageable objects. The import statement allows access to external packages and classes. It requires only one argument, which specifies the uniform resource identifier (URI) of the class/library. If the library is created by a package manager, then you specify the package: scheme before the URI. If importing a class, you specify the location and class name or the package: directive.

```
// Import the material package
import 'package:flutter/material.dart';

//          Import
external    class
import
'charts.dart';

// Import external class in a different folder
import 'services/charts_api.dart';

// Import external class with package: directive
import 'package:project_name/services/charts_api.dart';
```

USING CLASSES

All classes descend from Object, the base class for all Dart objects. A class has members (variables and methods) and uses a constructor to create an object. If a constructor is not declared, a default construc- tor will be provided automatically. The default constructor provided for you has no arguments.

What is a constructor, and why is it needed? A constructor has the same name as the class, with optional parameters. The parameters serve as getters of values when initializing a class for the first time. Dart uses syntactic sugar to make it easy to access values by using the this keyword, referring to

the current state in the class.

```
// Getter
this.type
= type;

// Syntactic
Sugar
this.type;
```

A basic class with a constructor would have this simple layout:

```
class Fruit
    { String
    type;

    // Constructor - Same name as class
    Fruit(this.type);
}
```

The previous example uses a constructor with syntactic sugar, Fruit(this.type), and the construc- tor is called in this manner: Fruit = Fruit('apple');. To use named parameters, enclose the parameter in curly brackets, Fruit({this.type}), and call the constructor in this manner: Fruit = Fruit(type: 'Apple');. Imagine passing three or four parameters; I prefer to use named param- eters to keep the code readable. Each parameter is optional unless you specify with @requiredthat it is a required parameter.

```
// Required parameter
Fruit({@required this.type});

// Constructor - With optional parameter name at
init Fruit({this.type});
```

In addition to marking a parameter @required, you can add the assert statement to show an error if a value is missing. The assertstatement throws an error during development (debug) mode and has no effect in production code (release).

```
// Constructor - Required parameter plus
assert class Fruit {
```

```
  String type;

  Fruit({@required this.type}) : assert(type != null);
}

// Call the Fruit class
Fruit fruit = Fruit(type:
'Apple'); print('fruit.type:
${fruit.type}');
```

In a class, methods are functions that provide logic for an object. Methods can return a value or void
(no return value, empty).

```
// Method in the class
calculateFruitCalories() {
   // Logic to calculate calories
}
```

Let's look at one example of a class without a constructor and two examples of a class with a con- structor and a named constructor.

First let's look at creating a class that does not define a constructor and declares two variables to hold the barista's name and experience. Since the example does not declare a constructor, a default constructor without any arguments is provided for you. What does this mean? Inside the class, it's the same as if you had typed BaristaNoConstructor();, which is a default constructor without arguments. You create an instance of the class by declaring a BaristaNoConstructor baristaNo- Constructorvariable initialized with BaristaNoConstructor(), which is the default constructor provided for you. By taking the baristaNoConstructor variable, you can use the dot operator like baristaNoConstructor.experience and give it a value of 10.

```
// Declare Classes

// Class Default No Arguments
Constructor class
BaristaNoConstructor {
    String
    name;
```

```
    int
    experi
    ence;
}
    // Class Default No Arguments Constructor
    BaristaNoConstructor baristaNoConstructor = BaristaNoConstructor();
    baristaNoConstructor.experience = 10;
    print('baristaNoConstructor.experience: ${baristaNoConstructor.experience}');
    // baristaNoConstructor.experience: 10
```

Next let's look at creating a class that defines a constructor with named parameters that hold the barista name and experience. This example shows how to add a method whatIsTheExperience() that returns the class's experience variable value. You create an instance of the class by declaring a BaristaWithConstructor barista variable initialized with the BaristaWithConstructor(name: 'Sandy', experience: 10) constructor. The benefits are immediately obvious when creating a class with a constructor. You can initialize each class's variables by passing the values via the constructor.

You can still use the dot operator to modify any of the variables, such as barista.experience.

```
    // Class Named
    Constructor class
    BaristaWithConstructo
    r {
        String
        name;
        int
        experi
        ence;

        // Constructor - Named parameters by using { }
        BaristaWithConstructor({this.name, this.experience});

        // Method - return
        value int
        whatIsTheExperien
```

```
    ce() {
        return experience;
    }
}
```

```
// Class Named Constructor and return value
BaristaWithConstructor barista = BaristaWithConstructor(name: 'Sandy',
experience: 10); int experienceByProperty = barista.experience;
int experienceByFunction = barista.whatIsTheExperience();
print('experienceByProperty: $experienceByProperty');
print('experienceByFunction: $experienceByFunction');
// experienceByProperty: 10
// experienceByFunction: 10
```

Named constructors allow you to implement multiple constructors for a class and provide clear inten- tions of initialized data.

Now let's look at creating a class that defines a named constructor that holds the barista name and experience. In this example, you'll build upon the previous example and add a second constructor— to be precise, a named constructor. You declare it by using the class name, the dot operator, and the name of the constructor, like BaristaNamedConstructor.baristaDetails(name: 'Sandy', experience: 10), giving you a named constructor using named parameters. You can still use the dot operator to modify any of the variables, such as barista.experience.

```
// Class with additional named constructor class
BaristaNamedConstructor {
    Strin
    g
    name
    ; int
    experi
    ence;
// Constructor - Named parameters { }
BaristaNamedConstructor({this.name,
```

```
    this.experience});

    // Named constructor - baristaDetails - With named parameters
    BaristaNamedConstructor.baristaDetails({this.name, this.experience});
}

BaristaNamedConstructor barista = BaristaNamedConstructor.baristaDetails(name: 'Sandy', experience: 10);
print('barista.name: ${barista.name} - barista.experience: ${barista.experience}');
// barista.name: Sandy - barista.experience: 10
```

Class Inheritance

In programming, inheritance allows objects to share traits. To inherit from other classes, use the extends keyword. Use the super keyword to refer to the superclass (the parent class). Constructors are not inherited in the subclass.

In this example, you'll take the previous BaristaNamedConstructor class and use inheritance to create a new class that inherits the parent class traits. Declare a new class with the name BaristaIn- heritance using the extends keyword and the name of the class you are extending, which here is BaristaNamedConstructor. The inherited class constructor looks just a little bit different than the previous declarations; at the end of the constructor, you add a colon (:) and super(), referring to the superclass. When the BaristaInheritance class is initialized, it inherits the parent class traits, meaning it can access variables and methods (class functions) from BaristaNamedConstructor.

```
// Class inheritance
class BaristaInheritance extends BaristaNamedConstructor { int yearsOnTheJob;

  BaristaInheritance({this.yearsOnTheJob}) : super();
}

// Init Inherited Class
BaristaInheritance baristaInheritance = BaristaInheritance(yearsOnTheJob: 7);
// Assign Parent Class variable
baristaInheritance.name = 'Sandy';
```

```
print('baristaInheritance.yearsOnTheJob:
${baristaInheritance.yearsOnTheJob}');          print('baristaInheritance.name:
${baristaInheritance.name}');
```

Class Mixins

Mixins are used to add features to a class and allow you to reuse the class code in different classes. In other words the mixins allow you to access class code between unrelated classes. To use a mixin, you add the with keyword followed by one or more mixin names. Place the with keyword right after the class name declaration. The class that implements a mixin does not declare a constructor. Usually the mixin class is a collection of methods. In Chapter 7, "Adding Animation to an App," you'll create two animation apps that use mixins. For example, using AnimationController relies on TickerProvider-StateMixin.
are retrieved, it will update the UI with the new data. In Chapter 13, "Local Persistence: Saving Data," you'll look at how to use Stream objects, which allow data to be added or returned in the future. To accomplish this, Dart uses StreamController and Stream.

The async and await keywords are used in conjunction. Mark the function as async and place the await keyword before the function that will return data in the future. Note that functions marked async must have a return type assignable to Future.

In this example, the totalCookiesCount() function implements a Future object that returns an int value. To implement a Future object, you start the function with Future<int> (int or any valid data type), the function name, and the async keyword. The code inside the function that returns a future value is marked with the await keyword. The lookupTotalCookiesCountDatabase() method represents a call to a web server to retrieve data and is preceded by the await keyword. The await keyword allows the request to be made, and instead of waiting for the data to come back, it continues executing the next block of code. Once the data is retrieved, the code continues to finish the function and returns the value.

```
// Async and await Function with Future - return value of integer
Future<int> totalCookiesCount() async {
```

```
    int cookiesCount = await lookupTotalCookiesCountDatabase(); //
    Returns 33 return cookiesCount;
}

// Async method to call web server
Future<int> lookupTotalCookiesCountDatabase() async {
  // In a real world app we call the web server to retrieve live data
  return 33;
}

//          User
pressed
button
totalCookies
Count()
     .then((count)              {
       print('cookiesCount:
       ${count}');
   });
print('This will print before cookiesCount');
// This will print before cookiesCount
// cookiesCount: 33
```

- **WHAT YOU LEARNED IN THIS Lab**

TOPIC	KEY CONCEPTS
Commenting the code	There are single-line, multiline, and documentation comments.
Accessing main()	main()is the top-level function.
Using variables	You can store values such as numbers, strings, Booleans, lists, maps, and runes.
Using operators	An operator is a symbol used to perform arithmetic, equality,

	relational, type test, assignment, logical, conditional, and cascade notation.
Using flow statements	Flow statements include if and else, the ternary operator, for loops, while and do-while, while and break, continue, and switch and case.
Using functions	Functions are used to group reusable logic.
Importing packages	You can use the import statement to import external packages, libraries or classes.
Using classes	You can create classes to separate code logic.

Lab Tasks

Exercise 1

Create a program that asks the user to enter their name and their age. Print out a message that tells how many years they have to be 100 years old.

Exercise 2

Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user.

Exercise 3

Take a list, say for example this one:

a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

and write a program that prints out all the elements of the list that are less than 5.

Exercise 4

Create a program that asks the user for a number and then prints out a list of all the divisors of that number.

If you don't know what a divisor is, it is a number that divides evenly into another number. For example, 13 is a divisor of 26 because 26 / 13 has no remainder.

Exercise 5

Take two lists, for example:

a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

and write a program that returns a list that contains only the elements that are common between them (without duplicates). Make sure your program works on two lists of different sizes.

Exercise 6

Ask the user for a string and print out whether this string is a palindrome or not.

A palindrome is a string that reads the same forwards and backwards.

Exercise 7

Let's say you are given a list saved in a variable:

a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
Write a Dart code that takes this list and makes a new list that has only the even elements of this list in it.

Exercise 8

Make a two-player Rock-Paper-Scissors game against computer.
Ask for player's input, compare them, print out a message to the winner.

Exercise 9

Generate a random number between 1 and 100. Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right.
Keep track of how many guesses the user has taken, and when the game ends, print this out.

Exercise 10

Ask the user for a number and determine whether the number is prime or not.
Do it using a function

Exercise 11

Write a program (function) that takes a list and returns a new list that contains all the elements of the first list minus all the duplicates.

Exercise 12

Write a password generator in Dart. Be creative with how you generate passwords - strong passwords have a mix of lowercase letters, uppercase letters, numbers, and symbols. The passwords should be random, generating a new password every time the user asks for a new password. Include your run-time code in a main method.
Ask the user how strong they want their password to be. For weak passwords, pick a word or two from a list.

Exercise 13

Create a program that will play the "cows and bulls" game with the user. The game works like this:

- Randomly generate a 4-digit number. Ask the user to guess a 4-digit number. For every digit the user guessed correctly in the correct place, they have a "cow". For every digit the user guessed correctly in the wrong place is a "bull."
- Every time the user makes a guess, tell them how many "cows" and "bulls" they have.

Once the user guesses the correct number, the game is over. Keep track of the number of guesses the user makes throughout the game and tell the user at the end.

Exercise 14

Time for some fake graphics! Let's say we want to draw game boards that look like this:

```
 --- --- ---
|  |   |   |
 --- --- ---
|  |   |   |
 --- --- ---
|  |   |   |
 --- --- ---
```

This one is 3x3 (like in tic tac toe).
Ask the user what size game board they want to draw, and draw it for them to the screen using Dart's print statement.

Exercise 15

You, the user, will have in your head a number between 0 and 100. The program will guess a number, and you, the user, will say whether it is too high, too low, or your number.
At the end of this exchange, your program should print out how many guesses it took to get your number.

Exercise 16

For this exercise, we will keep track of when our friend's birthdays are, and be able to find that information based on their name.
Create a dictionary (in your file) of names and birthdays. When you run your program it should ask the user to enter a name, and return the birthday of that person back to them. The interaction should look something like this:

```
>>> Welcome to the birthday dictionary. We know the birthdays of:
Albert Einstein
Benjamin Franklin
Ada Lovelace
>>> Who's birthday do you want to look up?
Benjamin Franklin
>>> Benjamin Franklin's birthday is 01/17/1706
```