



Department of Computer Science and Engineering

Guru Nanak Dev Engineering College, Ludhiana

Summer Training Report

App Development with Flutter

Submitted by:

Priyanka Jhamb

B.Tech 2nd year

U.R.N:1905379

Under the guidance of

Er. Ishant Kumar

Ed-Tech Veteran, Full-Stack-Development Expert, and an Entrepreneur



AURIBISES TECHNOLOGIES

2144, B20, Krishna Nagar Rd, Krishna Nagar, Ludhiana, Punjab 141002

Summer Training 2021

Declaration

I, Priyanka Jhamb, of the Department of Computer Science and Engineering, Guru Nanak Dev Engineering College, Ludhiana, hereby certify that the work which is being presented is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Priyanka Jhamb

Acknowledgement

I am highly grateful to the Dr. Sehajpal Singh, Principal, Guru Nanak Dev Engineering College (GNDEC), Ludhiana, for providing this opportunity to carry out the four-weeks practical training.

The constant guidance and encouragement received from Dr. Parminder Singh, H.O.D.(CSE Department), GNDEC Ludhiana has been of great help in carrying out the project work and is acknowledged with reverential thanks.

I would also like to express my special thanks of gratitude to my teacher Er. Ishant Kumar who gave me the knowledge of practical industrial aspects related to app development with Flutter and helped me out in the entire project of Food Delivery App.

I express gratitude to other faculty members of Computer Science and Engineering Department of GNDEC for their intellectual support throughout the course of this work.

Finally, I am indebted to all whosoever have contributed in this report work.

Priyanka Jhamb

Certificate of Training



**AURIBISES
TECHNOLOGIES**

CERTIFICATE OF COMPLETION 2021

This certificate is hereby bestowed upon

Priyanka Jhamb

For exceptional performance that has led to the successful completion of the course
App Development with Flutter

Project : Loving Food
from 16 July 2021 to 14 September 2021

given under hand and seal on this, the
14 of September 2021 at Ludhiana, India



An ISO 9001 - 2015 Company



Director

Registration No
Ref: ATPL/CC/20210724-ADF-1512

Date: 14th September, 2021

Er. Ishant Kumar
(Project Mentor)

Abstract

Nowadays Food Delivery App is the main important app most of the people are using as it gives them the chance to just order desired food and get it at home without going anywhere. Food app Development plays an important role in modern life because UI of app helps to interact with Hotels to order the food at home. So, latestlly, Dart is the popular and very easy programming language used to code Flutter apps. Dart is another product by Google and released version 2.1, before Flutter, in November and Flutter is a free and open-source mobile UI framework created by Google and released in May 2017. Flutter allows us to create native mobile applications with only one code. It means that we can use one programming language and one codebase to create two different apps (IOS and Android). We have used Flutter SDKs to build high-performance, high-fidelity apps for iOS, Android, Web, and Desktop from a single codebase. Flutter is all about 'Widgets' – a widget can be a Structural element like a button or menu, it can be a Stylistic element like a font or color, or a layout among many other things. Widgets are the basic building blocks of a Flutter app and it is an immutable declaration of part of the UI. Through learning and applying the knowledge, I have made some small interesting projects as well as 1 large project that is Food Delivery App.

Contents

1	Introduction	1
1.1	Dart	1
1.2	Flutter	1
2	Tour to Dart Programming Language	2
2.1	Language sample	2
2.1.1	Hello World	2
2.2	Comments	2
2.2.1	Single Line Comments	2
2.2.2	Multi-Line Comments	2
2.3	Return Type	2
2.4	Declaring variables	2
2.5	Default value	3
2.6	Assert Statements	3
2.7	Late variables	3
2.8	Final and const	4
2.9	Built-in types	4
2.9.1	Numbers	5
2.10	Functions	9
2.10.1	Parameters	9
2.10.2	main()	11
2.10.3	Functions as first-class objects	11
2.11	Operators	11
2.12	Control flow statements	12
2.13	Classes	14
2.13.1	Using class members	14
2.13.2	Getting an object's type	15
2.13.3	Instance variables	15
2.13.4	Constructors	16
2.13.5	Getters and setters	17
2.13.6	Abstract methods	17
2.13.7	Abstract classes	18
2.13.8	Extending a class	18
2.13.9	Overriding members	19
2.13.10	Adding features to a class: mixins	19
2.13.11	Class variables and methods	20
2.14	Some Important Cases	21
2.14.1	Related to Variables initialisation or assigning and Unicode Character Tables	21
2.14.2	toString function, Runtime type, Initialisation before Use Principe and using ? operator	23

2.14.3	final and const	24
2.14.4	Difference between exit and return & Named parameters	25
2.14.5	Abstract Class Real Life Example	25
2.14.6	Use of 'dart:math' and generate random string	27
2.15	Assignment	27
2.15.1	A Program For Water Bottle Sensor using basic variables and control function	27
2.15.2	A Program to sort the covid cases according to the Active, Confirmed, Recovered, Deceased and Tested	29
2.15.3	A Program for small case study of website makemytrip using inheritance and classes concept	33
3	Flutter	36
3.1	Flutter Install	36
3.1.1	System Requirements in Linux	36
3.1.2	Get the Flutter SDK	36
3.1.3	Run flutter doctor	36
3.1.4	Update your path	37
3.2	Android setup	37
3.3	Create the app	37
3.4	Write your first Flutter app	38
4	Work Done	42
4.1	Experimental setup	42
4.1.1	42
5	Future Work	43
5.1	43
5.1.1	43
6	Conclusion	44
	References	45

Chapter 1

Introduction

1.1 Dart

Dart is the programming language used to code Flutter apps. Dart is another product by Google and released version 2.1, before Flutter, in November. It is designed for client development, such as for the web and mobile apps. It can also be used to build server and desktop applications.

Dart is a client-optimized language for developing fast apps on any platform. Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks.

Dart offers sound null safety, meaning that values can't be null unless you say they can be. With sound null safety, Dart can protect you from null exceptions at runtime through static code analysis. Unlike many other null-safe languages, when Dart determines that a variable is non-nullable, that variable is always non-nullable. Dart has a rich set of core libraries, providing essentials for many everyday programming tasks.

Dart programming language is a completely object-oriented language, including basic types such as int variables are also objects.

1.2 Flutter

Google had its first ever release of Flutter 1.0 last December, after having it in beta mode for over 18 months. Flutter is Google's portable UI toolkit for crafting beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source. Flutter primarily uses Dart (a language also developed by google).

Chapter 2

Tour to Dart Programming Language

2.1 Language sample

2.1.1 Hello World

```
void main() {  
    print('Hello , World!');  
}
```

2.2 Comments

2.2.1 Single Line Comments

```
// This is a single line comment.
```

2.2.2 Multi-Line Comments

```
/* This is a multi-line comment.*/
```

2.3 Return Type

void: A special type that indicates a value that's never used. Functions like `printInteger()` and `main()` that don't explicitly return a value have the `void` return type.

int: It indicates that function will return integer value.

Similarly, with the `String`, `Bool` Return type etc.

2.4 Declaring variables

var: A way to declare a variable without specifying its type. The type of this variable (`int`) is determined by its initial value .

```
var name = 'Bob';
```

Variables store references. The variable called `name` contains a reference to a `String` object with a value of `"Bob"`.

The type of the name variable is inferred to be String, but you can change that type by specifying it. If an object isn't restricted to a single type, specify the Object type (or dynamic if necessary).

```
Object name = 'Bob';
```

Another option is to explicitly declare the type that would be inferred:

```
String name = 'Bob';
```

2.5 Default value

Uninitialized variables that have a nullable type have an initial value of null. (If you haven't opted into null safety, then every variable has a nullable type.) Even variables with numeric types are initially null, because numbers-like everything else in Dart-are objects.

```
int? lineCount;
```

If you enable null safety, then you must initialize the values of non-nullable variables before you use them:

```
int lineCount = 0;
```

You don't have to initialize a local variable where it's declared, but you do need to assign it a value before it's used.

2.6 Assert Statements

As a programmer, it is very necessary to make an errorless code is very necessary and to find the error is very difficult in a big program. Dart provides the programmer with assert statements to check for the error. The assert statement is a useful tool to debug the code and it uses boolean condition for testing. If the boolean expression in assert statement is true then the code continues to execute, but if it returns false then the code ends with Assertion.

Syntax:

```
assert(condition);
```

It must be noted that if you want to use assert then you have to enable it while execution as it can only be used in the development mode and not in productive mode. If it is not enabled then it will be simply be ignored while execution.

Enable the assert while executing a dart file via cmd as:

```
dart --enable-asserts file_name.dart
```

2.7 Late variables

If you're sure that a variable is set before it's used, but Dart disagrees, you can fix the error by marking the variable as late:

```
late String description;  
  
void main() {  
  description = 'Feijoada!';  
}
```

```
    print(description);  
}
```

Warning: If you fail to initialize a late variable, a runtime error occurs when the variable is used.

If you fail to initialize a late variable, a runtime error occurs when the variable is used. When you mark a variable as late but initialize it at its declaration, then the initializer runs the first time the variable is used. This lazy initialization is handy in a couple of cases:

- The variable might not be needed, and initializing it is costly.
- You're initializing an instance variable, and its initializer needs access to this.

In the following example, if the temperature variable is never used, then the expensive `readThermometer()` function is never called:

```
// This is the program's only call to _readThermometer().  
late String temperature = readThermometer(); // Lazily initialized.
```

2.8 Final and const

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.)

Instance variables can be `final` but not `const`.

Here's an example of creating and setting a final variable:

```
final name = 'Bob'; // Without a type annotation  
final String nickname = 'Bobby';  
name = 'Alice'; // Error: a final variable can only be set once.
```

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`. Where you declare the variable, set the value to a compile-time constant such as a number or string literal, a `const` variable, or the result of an arithmetic operation on constant numbers:

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

You can change the value of a non-final, non-const variable, even if it used to have a `const` value:

```
foo = [1, 2, 3]; // Was const []
```

2.9 Built-in types

The Dart language has special support for the following:

- Numbers (`int`, `double`)
- Strings (`String`)
- Booleans (`bool`)

- Lists (List, also known as arrays)
- Sets (Set)
- Maps (Map)
- Runes (Runes; often replaced by the characters API)
- Symbols (Symbol)
- The value null (Null)

This support includes the ability to create objects using literals. For example, 'this is a string' is a string literal, and true is a boolean literal.

Because every variable in Dart refers to an object—an instance of a class—you can usually use constructors to initialize variables. Some of the built-in types have their own constructors. For example, you can use the Map() constructor to create a map.

2.9.1 Numbers

- int: Integer values no larger than 64 bits, depending on the platform.
- double: 64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard.
- Strings: A Dart string (String object) holds a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```
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.

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
'Dart has string interpolation, '
'which is very handy.');
```

```
assert('That deserves all caps. '
'$s.toUpperCase() is very handy!' ==
'That deserves all caps. '
'STRING INTERPOLATION is very handy!');
```

Note: The == operator tests whether two objects are equivalent. Two strings are equivalent if they contain the same sequence of code units.

You can concatenate strings using adjacent string literals or the + operator:

```
var s1 = 'String '
'concatenation '
" works even over line breaks.";
assert(s1 ==
'String concatenation works even over '
'line breaks.');
```

```
var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

Another way to create a multi-line string: use a triple quote with either single or double quotation marks:

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

```
var s2 = """This is also a
multi-line string.""";
```

You can create a `raw` string by prefixing it with `r`:

```
var s = r'In a raw string, not even \n gets special treatm
```

- **Booleans:** To represent boolean values, Dart has a type named `bool`. Only two objects have type `bool`: the boolean literals `true` and `false`, which are both compile-time constants.

Dart's type safety means that you can't use code like `if (nonbooleanValue)` or `assert (nonbooleanValue)`. Instead, explicitly check for values, like this:

```
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty);
```

```
// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);
```

```
// Check for null.
var unicorn;
assert(unicorn == null);
```

```
// Check for NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

- **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, so most people just call them lists.

Here's a simple Dart list:

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

Lists use zero-based indexing, where 0 is the index of the first value and `list.length - 1` is the index of the last value.

```
var list = [1, 2, 3];
assert(list.length == 3);
assert(list[1] == 2);

list[1] = 1;
assert(list[1] == 1);
```

To create a list that's a compile-time constant, add `const` before the list literal:

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // This line will cause an error.
```

Spread Operator: Dart 2.3 introduced the spread operator (`...`) and the null-aware spread operator (`...?`), which provide a concise way to insert multiple values into a collection.

For example, you can use the spread operator (`...`) to insert all the values of a list into another list:

```
var list = [1, 2, 3];
var list2 = [0, ...list];
assert(list2.length == 4);
```

Null aware Spread Operator: If the expression to the right of the spread operator might be null, you can avoid exceptions by using a null-aware spread operator (`...?`):

```
var list;
var list2 = [0, ...? list];
assert(list2.length == 1);
```

Collection if and Collection for: Dart also offers collection if and collection for, which you can use to build collections using conditionals (if) and repetition (for).

Here's an example of using collection if to create a list with three or four items in it:

```
void main() {
  bool promo=false;
  var nav = [
    'Home',
    'Furniture',
    'Plants',
    if (promo) 'Outlet'
  ];
  print(nav);
}
```

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

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

- **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. Here is a simple Dart set, created using a set literal:

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

Note: Dart infers that halogens has the type Set<String>. If you try to add the wrong type of value to the set, the analyzer or runtime raises an error.

Empty Set: To create an empty set, use `Set<String>{};` preceded by a type argument, or `var names = {};` assign to a variable of type Set:

```
var names = <String>{};
// Set<String> names = {}; // This works, too.
// var names = {}; // Creates a map, not a set.
```

Set or map? The syntax for map literals is similar to that for set literals. Because map literals came first, defaults to the Map type. If you forget the type annotation on `names` or the variable it's assigned to, then Dart creates an object of type Map<dynamic, dynamic>.

Add items to an existing set using the `add()` or `addAll()` methods:

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

- **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. Dart support for maps is provided by map literals and the Map type.

Here are a couple of simple Dart maps, created using map literals:

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

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

```
};
```

You can create the same objects using a Map constructor:

```
var gifts = Map<String, String>();  
gifts['first'] = 'partridge';  
gifts['second'] = 'turtledoves';  
gifts['fifth'] = 'golden rings';  
  
var nobleGases = Map<int, String>();  
nobleGases[2] = 'helium';  
nobleGases[10] = 'neon';  
nobleGases[18] = 'argon';
```

2.10 Functions

Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it can be used over and over and over again.

Here's an example of implementing a function:

```
bool isNoble(int atomicNumber) {  
    return _nobleGases[atomicNumber] != null;  
}
```

Although Effective Dart recommends type annotations for public APIs, the function still works if you omit the types:

```
isNoble(atomicNumber) {  
    return _nobleGases[atomicNumber] != null;  
}
```

For functions that contain just one expression, you can use a shorthand syntax:

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

2.10.1 Parameters

A function can have any number of required positional parameters. These can be followed either by named parameters or by optional positional parameters (but not both).

Named parameters

Named parameters are optional unless they're specifically marked as required.

When calling a function, you can specify named parameters using `paramName: value`. For example:

```
enableFlags(bold: true, hidden: false);
```

When defining a function, use `param1`, `param2`, etc to specify named parameters:


```
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool? bold, bool? hidden}) {...}
```

Although named parameters are a kind of optional parameter, you can annotate them with `required` to indicate that the parameter is mandatory – that users must provide a value for the parameter. For example:

```
const Scrollbar({Key? key, required Widget child})
```

Optional positional parameters

Wrapping a set of function parameters in `[]` marks them as optional positional parameters:

```
String say(String from, String msg, [String? device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Here's an example of calling this function without the optional parameter:

```
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

And here's an example of calling this function with the third parameter:

```
assert(say('Bob', 'Howdy', 'smoke signal') ==
  'Bob says Howdy with a smoke signal');
```

Default parameter values

Your function can use `=` to define default values for both named and positional parameters. The default values must be compile-time constants. If no default value is provided, the default value is `null`.

Here's an example of setting default values for named parameters:

```
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool bold = false, bool hidden = false}) {...}

// bold will be true; hidden will be false.
enableFlags(bold: true);
```

The next example shows how to set default values for positional parameters:

```
String say(String from, String msg,
  [String device = 'carrier pigeon']) {
  var result = '$from says $msg with a $device';
  return result;
}

assert(say('Bob', 'Howdy') ==
  'Bob says Howdy with a carrier pigeon');
```

2.10.2 main()

Every app must have a top-level `main()` function, which serves as the entrypoint to the app. The `main()` function returns `void` and has an optional `List<String>` parameter for arguments. Here's a simple `main()` function:

```
void main() {  
    print('Hello, World!');  
}
```

Here's an example of the `main()` function for a command-line app that takes arguments:

```
// Run the app like this: dart args.dart 1 test  
void main(List<String> arguments) {  
    print(arguments);  
  
    assert(arguments.length == 2);  
    assert(int.parse(arguments[0]) == 1);  
    assert(arguments[1] == 'test');  
}
```

2.10.3 Functions as first-class objects

You can pass a function as a parameter to another function. For example:

```
cvoid printElement(int element) {  
    print(element);  
}  
  
var list = [1, 2, 3];  
  
// Pass printElement as a parameter.  
list.forEach(printElement);
```

You can also assign a function to a variable, such as:

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';  
assert(loudify('hello') == '!!! HELLO !!!');
```

2.11 Operators

Dart supports the operators shown in the following table.

Description	Operator
unary postfix	expr++ expr-- () [] ?[] . ?.
unary prefix	-expr !expr expr ++expr --expr await 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	.. ?..
assignment	= *= /= += -= &= ^= etc.

In the operator table, each operator has higher precedence than the operators in the rows that follow it. For example, the multiplicative operator % has higher precedence than (and thus executes before) the equality operator ==, which has higher precedence than the logical AND operator &&. That precedence means that the following two lines of code execute the same way:

```
// Parentheses improve readability.
if ((n % i == 0) && (d % i == 0)) ...

// Harder to read, but equivalent.
if (n % i == 0 && d % i == 0) ...
```

2.12 Control flow statements

- if and else: Dart supports if statements with optional else statements, as the next sample shows.

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

- for loops: You can iterate with the standard for loop. For example:

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

- while and do-while loops: A while loop evaluates the condition before the loop:

```
while (!isDone()) {
  doSomething();
}
```

A do-while loop evaluates the condition after the loop:

```
do {
  printLine();
} while (!atEndOfPage());
```

- break and continue: Use break to stop looping:

```
while (true) {
  if (shutdownRequested()) break;
  processIncomingRequests();
}
```

Use continue to skip to the next loop iteration:

```
for (int i = 0; i < candidates.length; i++) {
  var candidate = candidates[i];
  if (candidate.yearsExperience < 5) {
    continue;
  }
  candidate.interview();
}
```

You might write that example differently if you're using an Iterable such as a list or set:

```
candidates
  .where((c) => c.yearsExperience >= 5)
  .forEach((c) => c.interview());
```

- switch and case: Switch statements in Dart compare integer, string, or compile-time constants using ==. The compared objects must all be instances of the same class (and not of any of its subtypes), and the class must not override ==. Enumerated types work well in switch statements.

Each non-empty case clause ends with a break statement, as a rule. Other valid ways to end a non-empty case clause are a continue, throw, or return statement.

Use a default clause to execute code when no case clause matches:

```

var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}

```

However, Dart does support empty case clauses, allowing a form of fall-through:

- **assert:** During development, use an assert statement - `assert(condition, optionalMessage);` - to disrupt normal execution if a boolean condition is false.

2.13 Classes

Dart is an object-oriented language with classes and mixin-based inheritance. Every object is an instance of a class, and all classes except `Null` descend from `Object`. Mixin-based inheritance means that although every class (except for the top class, `Object`?) has exactly one superclass, a class body can be reused in multiple class hierarchies. Extension methods are a way to add functionality to a class without changing the class or creating a subclass.

2.13.1 Using class members

Objects have members consisting of functions and data (methods and instance variables, respectively). When you call a method, you invoke it on an object: the method has access to that object's functions and data.

Use a dot (`.`) to refer to an instance variable or method:

```

var p = Point(2, 2);

// Get the value of y.
assert(p.y == 2);

// Invoke distanceTo() on p.
double distance = p.distanceTo(Point(4, 4));

```

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null:

```
// If p is non-null, set a variable equal to its y value.
var a = p?.y;
```

Using constructors You can create an object using a constructor. Constructor names can be either `ClassName` or `ClassName.identifier`. For example, the following code creates `Point` objects using the `Point()` and `Point.fromJson()` constructors:

```
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

The following code has the same effect, but uses the optional `new` keyword before the constructor name:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

2.13.2 Getting an object's type

To get an object's type at runtime, you can use the `Object` property `runtimeType`, which returns a `Type` object.

```
print('The type of a is ${a.runtimeType}');
```

2.13.3 Instance variables

Here's how you declare instance variables:

```
class Point {
  double? x; // Declare instance variable x, initially null.
  double? y; // Declare y, initially null.
  double z = 0; // Declare z, initially 0.
}
```

All uninitialized instance variables have the value `null`.

All instance variables generate an implicit getter method. Non-final instance variables and late final instance variables without initializers also generate an implicit setter method. For details, see [Getters and setters](#).

If you initialize a non-late instance variable where it's declared, the value is set when the instance is created, which is before the constructor and its initializer list execute.

```
class Point {
  double? x; // Declare instance variable x, initially null.
  double? y; // Declare y, initially null.
}

void main() {
  var point = Point();
  point.x = 4; // Use the setter method for x.
  assert(point.x == 4); // Use the getter method for x.
  assert(point.y == null); // Values default to null.
}
```

Instance variables can be `final`, in which case they must be set exactly once. Initialize final, non-late instance variables at declaration, using a constructor parameter, or using a constructor's initializer list:

```
class ProfileMark {
  final String name;
  final DateTime start = DateTime.now();

  ProfileMark(this.name);
  ProfileMark.unnamed() : name = '';
}
```

2.13.4 Constructors

Declare a constructor by creating a function with the same name as its class (plus, optionally, an additional identifier as described in Named constructors). The most common form of constructor, the generative constructor, creates a new instance of a class:

```
class Point {
  double x = 0;
  double y = 0;

  Point(double x, double y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

The `this` keyword refers to the current instance.

Note: Use `this` only when there is a name conflict. Otherwise, Dart style omits the `this`. The pattern of assigning a constructor argument to an instance variable is so common, Dart has syntactic sugar to make it easy:

```
class Point {
  double x = 0;
  double y = 0;

  // Syntactic sugar for setting x and y
  // before the constructor body runs.
  Point(this.x, this.y);
}
```

Default constructors

If you don't declare a constructor, a default constructor is provided for you. The default constructor has no arguments and invokes the no-argument constructor in the superclass.

Constructors aren't inherited

Subclasses don't inherit constructors from their superclass. A subclass that declares no constructors has only the default (no argument, no name) constructor.

Named constructors

Use a named constructor to implement multiple constructors for a class or to provide extra clarity:

```

const double xOrigin = 0;
const double yOrigin = 0;

class Point {
    double x = 0;
    double y = 0;

    Point(this.x, this.y);

    // Named constructor
    Point.origin()
    : x = xOrigin,
      y = yOrigin;
}

```

Remember that constructors are not inherited, which means that a superclass's named constructor is not inherited by a subclass. If you want a subclass to be created with a named constructor defined in the superclass, you must implement that constructor in the subclass.

2.13.5 Getters and setters

Getters and setters are special methods that provide read and write access to an object's properties. Recall that each instance variable has an implicit getter, plus a setter if appropriate. You can create additional properties by implementing getters and setters, using the `get` and `set` keywords:

```

class Rectangle {
    double left, top, width, height;

    Rectangle(this.left, this.top, this.width, this.height);

    // Define two calculated properties: right and bottom.
    double get right => left + width;
    set right(double value) => left = value - width;
    double get bottom => top + height;
    set bottom(double value) => top = value - height;
}

void main() {
    var rect = Rectangle(3, 4, 20, 15);
    assert(rect.left == 3);
    rect.right = 12;
    assert(rect.left == -8);
}

```

With getters and setters, you can start with instance variables, later wrapping them with methods, all without changing client code.

2.13.6 Abstract methods

Instance, getter, and setter methods can be abstract, defining an interface but leaving its implementation up to other classes. Abstract methods can only exist in abstract classes.

To make a method abstract, use a semicolon (;) instead of a method body:


```

abstract class Doer {
    // Define instance variables and methods...

    void doSomething(); // Define an abstract method.
}

class EffectiveDoer extends Doer {
    void doSomething() {
        // Provide an implementation, so the method is not abstract here
        ...
    }
}

```

2.13.7 Abstract classes

Use the abstract modifier to define an abstract class—a class that can't be instantiated. Abstract classes are useful for defining interfaces, often with some implementation. If you want your abstract class to appear to be instantiable, define a factory constructor.

Abstract classes often have abstract methods. Here's an example of declaring an abstract class that has an abstract method:

```

// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
    // Define constructors, fields, methods...

    void updateChildren(); // Abstract method.
}

```

2.13.8 Extending a class

Use extends to create a subclass, and super to refer to the superclass:

```

class Television {
    void turnOn() {
        _illuminateDisplay();
        _activateIrSensor();
    }
    // ...
}

class SmartTelevision extends Television {
    void turnOn() {
        super.turnOn();
        _bootNetworkInterface();
        _initializeMemory();
        _upgradeApps();
    }
    // ...
}

```

2.13.9 Overriding members

Subclasses can override instance methods (including operators), getters, and setters. You can use the `@override` annotation to indicate that you are intentionally overriding a member:

```
class Television {  
    // ÆÛÆÛÆÛ  
    set contrast(int value) {...}  
}  
  
class SmartTelevision extends Television {  
    @override  
    set contrast(num value) {...}  
    // ÆÛÆÛÆÛ  
}
```

An overriding method declaration must match the method (or methods) that it overrides in several ways:

The return type must be the same type as (or a subtype of) the overridden method's return type. Argument types must be the same type as (or a supertype of) the overridden method's argument types. In the preceding example, the contrast setter of SmartTelevision changes the argument type from `int` to a supertype, `num`. If the overridden method accepts `n` positional parameters, then the overriding method must also accept `n` positional parameters. A generic method can't override a non-generic one, and a non-generic method can't override a generic one.

2.13.10 Adding features to a class: mixins

Mixins are a way of reusing a class's code in multiple class hierarchies.

To use a mixin, use the `with` keyword followed by one or more mixin names. To implement a mixin, create a class that extends `Object` and declares no constructors. Unless you want your mixin to be usable as a regular class, use the `mixin` keyword instead of `class`. For example:

```
mixin c {  
    int g=0;  
    void f()  
    {  
        print('Hello');  
    }  
}  
mixin b  
{  
    int d=0;  
    void u()  
    {  
        print("World");  
    }  
}  
class a with c,b  
{  
    void func()  
    {
```

```

        f();
        u();
        print(g);
        print(d);

    }
}
void main()
{
    a obj=a();
    obj.g=78;
    obj.func();
}

```

Sometimes you might want to restrict the types that can use a mixin. For example, the mixin might depend on being able to invoke a method that the mixin doesn't define. As the following example shows, you can restrict a mixin's use by using the `on` keyword to specify the required superclass:

```

class Musician {
    // ...
}
mixin MusicalPerformer on Musician {
    // ...
}
class SingerDancer extends Musician with MusicalPerformer {
    // ...
}

```

In the preceding code, only classes that extend or implement the `Musician` class can use the mixin `MusicalPerformer`. Because `SingerDancer` extends `Musician`, `SingerDancer` can mix in `MusicalPerformer`.

2.13.11 Class variables and methods

Use the `static` keyword to implement class-wide variables and methods.

Static variables (class variables)

Static variables (class variables) are useful for class-wide state and constants:

```

class Queue {
    static const initialCapacity = 16;
    // ...
}

void main() {
    assert(Queue.initialCapacity == 16);
}

```

Static variables aren't initialized until they're used.

Static methods

Static methods (class methods) don't operate on an instance, and thus don't have access to this. They do, however, have access to static variables. As the following example shows, you invoke static methods directly on a class:

```
import 'dart:math';

class Point {
  double x, y;
  Point(this.x, this.y);

  static double distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

2.14 Some Important Cases

2.14.1 Related to Variables initialisation or assigning and Unicode Character Tables

```
void main()
{
  //simple printing the statement
  print("hello");

  // int i;
  // print("${i}");//error comes as initialization has not there.

  int j=9;
  print("${j}");

  j=8;
  print("${j}");

  String Q="tr";
  // Q=3; error comes

  // j=3.3;
  // print("${j}");//error comes as we cannot assign float value to
  int value.
```

```

double k=8.8;
print("${k}");

k=3;
print("${k}");//can assign int value to double value as memory
               consumed by double value more than int value.. so type casting has
               been done.

//var

//Case 1
var m=3;
print("${m}");

// m=5.2; //error comes
// print("${m}");

//Case 2
var n=3.6;
print("${n}");

n=5;
print("${n}");

//Case 3
var p=3.6;
print("${p}");

// p="priya";//error comes
// print("${p}");

//Object

Object i;

i="Priya";
print("${i}");
i=9;
print("${i}");
i=8.7;
print("${i}");
i=true;
print("${i}");

//Unicode Character Table: https://unicode-table.com/en/

print("\u20b9");
print("ð$ŸÀ");

// Exploratory -> Samrt Water Bottle
// Explore - Operators Once Again

```

```
}
```

```
PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE
Connecting to VM Service at http://127.0.0.1:36095/10F5z6fGgBs=/
hello
9
8
8.8
3.0
3
3.6
5.0
3.6
Priya
9
8.7
true
₹
😄
Exited
```

2.14.2 toString function, RuntimeType, Initialisation before Use Principle and using ? operator

```
void main()
{
// variable.runtimeType()
var age=340000000098768978;// age is a reference variable which will
    always hold hashcodes
print("variable age is of the type of: ${age.runtimeType}");
print("age hashCode is: ${age.hashCode}");

// Initialization Before Use Principle
int? y;//It contains null value till we don't assign any value
print("y is ${y}");
print(y.toString());
//print(y!.toString());//! is a safety check due to which this
    statement only works when value of y is not null. ( execute
    toString if name is not null)
```

```

//Difference between these two statements
//String name = ""; //Due to this some memory will allocate to the
//variable name
//String? name; //Due to this, memory will be saved as it will not
//allocate till we don't assign value.

// variable.isEmpty()
String name="";
//String? name; //using, this error comes
if (name.isEmpty)
{
    print("Firstly fill the name.");
}

// Lazy Initialization
late String response;
//response = getNewsFromAPI();
response="smartwork";
print("name is: ${response}");
}

```

```

PROBLEMS 37 OUTPUT TERMINAL DEBUG CONSOLE
variable age is of the type of: int
age hashCode is: 340000000098768978
y is null
null
Firstly fill the name.
name is: smartwork
Exited

```

2.14.3 final and const

Assignment1CovidCasesTabularform.dart

```

//const and final

const x=45;
void main()
{
    // const is generally used in functions i.e. local scope
    // or global scope
    const y=3;
    const z=y+3;
    print("${z}");
    // final marks the instance variables i.e. attributes of an object as
    final
}

```

```
final abc=454;
print("${abc}");
}
```

2.14.4 Difference between exit and return & Named parameters

```
import 'dart:io';
printNumber({num:5})
{
  //first method
  print(num);
  if (num>1)
  {
    printNumber(num:num-1);
  }
  // exit means exit from app,
  // but return means exit from only function.

  //second method

  // if (num==1)
  // {
  //   return; //exit(0);
  // }
  // else
  // {
  //   printNumber(num: num-1);
  // }
  // if(num>0){printNumber(num: num-1);}
  // else{return;}
}
void main()
{
  printNumber(num:10);
}
```

2.14.5 Abstract Class Real Life Example

```
abstract class Notification{
  void onNotification(String message);
}

class YoutubeChannel{

  Notification? notification;

  void subscribe(User user){
    notification = user; // Polymorphism
    // notification is a reference which is now pointing to the object
    of User
  }
}
```



```

void uploadNewVideo(String title){
    print("Upload Started ....");
    print("Upload Finished ....");
    notification!.onNotification("A new Video has been uploaded ${
title}");
}
}

class User extends Notification{

    String? name;
    String? email;

    User({this.name, this.email});

    @override
    String toString() {
        return "${name} | ${email}";
    }

    void onNotification(String message){
        print("~~~~~");
        print("New Notification");
        print(message);
        print("~~~~~");
    }
}

void main(){

    YoutubeChannel channel = YoutubeChannel();

    User user1 = User(name: "John", email: "john@example.com");
    channel.subscribe(user1);

    channel.uploadNewVideo("Intro to OOPS :)");
}

```

PROBLEMS 37 OUTPUT TERMINAL DEBUG CONSOLE

```

Upload Started....
Upload Finished....

~~~~~
New Notification
A new Video has been uploaded Intro to OOPS :)

~~~~~

Exited

```

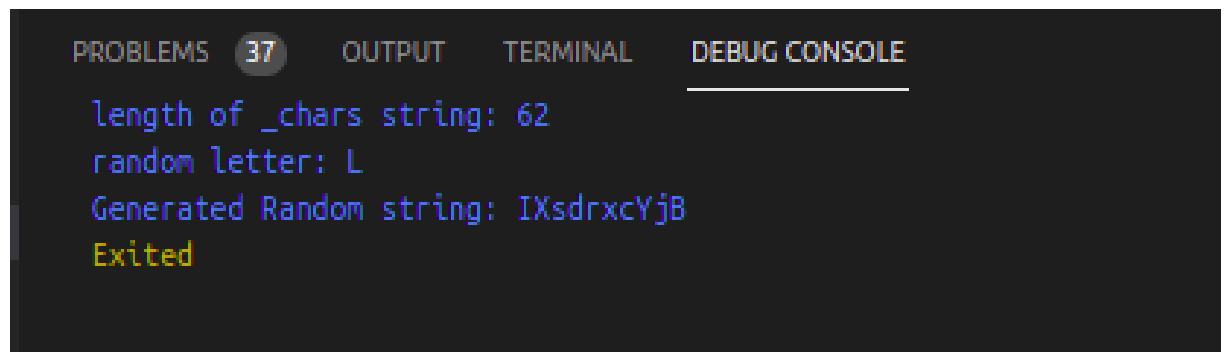
2.14.6 Use of 'dart:math' and generate random string

```
import 'dart:math';
var r = Random();
const _chars = 'AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz1234567890';

String generateRandomString(int len) {

    return List.generate(len, (index) => _chars[r.nextInt(_chars.length)]).join(); //generator function
}

void main() {
    print("length of _chars string: ${_chars.length}");
    print("random letter: ${_chars[r.nextInt(_chars.length)]}"); //
    // nextInt() takes a max int argument and generates a positive int
    // between 0 (inclusive) and max, exclusive.
    print("Generated Random string: ${generateRandomString(10)}");
}
```



```
PROBLEMS 37 OUTPUT TERMINAL DEBUG CONSOLE
length of _chars string: 62
random letter: L
Generated Random string: IXsdrxcYjB
Exited
```

2.15 Assignment

2.15.1 A Program For Water Bottle Sensor using basic variables and control function

```
// // Program for Water Bottle Sensor:
// It has 3 operations—>
// 1. Update how much water we drank.
// 2. Update if it is empty.
// 3. Update if it is refilled.
// need to maintain the quantity in all 3 secenarios.

void main() {
    //water measure in litres
    double threshold_value=7.5;
    double sensor=7.5;
    double water_drank= 6.5;
```

```

double water_fill=7.5;

//first scenario

// full
if (sensor==threshold_value)
{
    print("Water bottle is full");
}

//empty

if(sensor==0.0)
{
    print("water bottle is empty.");
}

//2nd scenario

if (water_drank<=sensor)
{
    sensor-=water_drank;
    print("${sensor} litre of water is left.");
}
else
{
    print("required water is not in the bottle");
}

//3rd scenario

if (water_fill>(threshold_value-sensor))
{
    print("Water is more that the water bottle can contain");
}
else
{
    sensor+=water_fill;
    print("${sensor} litre of water is in the bottle");
}
}

```

PROBLEMS 37 OUTPUT TERMINAL DEBUG CONSOLE

```

Water bottle is full
1.0 litre of water is left.
Water is more that the water bottle can contain
Exited

```

2.15.2 A Program to sort the covid cases according to the Active, Confirmed, Recovered, Deceased and Tested

```
// https://www.covid19india.org/
// CovidCases

import 'dart:io';
void main()
{
    Map Maharashtra={'Confirmed': 6229596, 'Active':94593,'Recovered'
        :6000911, 'Deceased':130753, 'Tested':5546565};
    Map Kerala={'Confirmed': 3187716, 'Active':126396,'Recovered'
        :3045310, 'Deceased':15512,'Tested':554665};
    Map Karnataka={'Confirmed': 2886702, 'Active':26256,'Recovered'
        :2824197, 'Deceased':36226,'Tested':56565};
    Map TamilNadu={'Confirmed': 2539277, 'Active':26717,'Recovered'
        :2478778, 'Deceased':33782,'Tested':554685};
    Map AndhraPradesh={'Confirmed': 1946749, 'Active':23939,'Recovered'
        :1909613, 'Deceased':13197,'Tested':554667565};
    Map UttarPradesh={'Confirmed': 1708005, 'Active':1036,'Recovered'
        :1684230, 'Deceased':22739,'Tested':553565};
    var states={'Maharashtra': Maharashtra, 'Kerala': Kerala, 'Karnataka'
        : Karnataka, 'TamilNadu':TamilNadu, 'AndhraPradesh':AndhraPradesh
        , 'UttarPradesh': UttarPradesh};

    List stateskeys=states.keys.toList();
    List statesvalues=states.values.toList();
    // print(stateskeys);
    // print(statesvalues);
    print("#"*155);
    print("\t\t\t\t\tStates with number of cases ");
    print("#"*155);

    List caseskeys=statesvalues[0].keys.toList();
    //formatting
    int max=0;
    for(int u=0;u<statesvalues.length;u++)
    {
        for(int w=0;w<caseskeys.length;w++)
        {
            if (max<statesvalues[u][caseskeys[w]].toString().length)
            {
                max=statesvalues[u][caseskeys[w]].toString().length;
            }
        }
    }
}
```

```

    }
}

int maxlength=0;
for(int v=0;v<caseskeys.length;v++)
{
    if (maxlength<caseskeys[v].length)
    {
        maxlength=caseskeys[v].length;
    }
}

if (max>maxlength)
{
    maxlength=max;
}
else if (maxlength>max)
{
    max=maxlength;
}

stdout.write("| \t States \t |");

for(int r=0;r<caseskeys.length;r++)
{
    stdout.write(" \t ${caseskeys[r]}");
    if ([caseskeys[r]].length<maxlength)
    {

        int temp4=caseskeys[r].length;
        int temp5=maxlength-temp4;

        for(int y=0;y<temp5;y++)
        {
            stdout.write(" ");
        }

    }
    stdout.write("\t|");
}

for(int i=0;i<statesvalues.length;i++)
{
    stdout.write("\n| \t ${stateskeys[i]} \t |");
    for(int s=0;s<caseskeys.length;s++)
    {
        stdout.write(" \t ${statesvalues[i][caseskeys[s]]}");
        if (statesvalues[i][caseskeys[s]].toString().length<max)
        {
            var temp9=statesvalues[i][caseskeys[s]].toString();
            var temp10=temp9.length;
            int temp3=max-temp10;

```

```

        for(int y=0;y<temp3;y++)
        {
            stdout.write(" ");
        }
    }
    stdout.write("\t|");

}

}
print("\n\n");
print("*"*155);
print("Types of cases are : ${caseskeys}");
for(int p=0;p<caseskeys.length;p++)
{
    print("${caseskeys[p]}: ${p}");
}

print("put variable typeofcases value to 0/1/2/3 according to what
you need for sorting according to that cases.");

//Sorting starts
int typeofcases=1;
print("for the time being , typeofcases=${typeofcases}—> ${caseskeys
[typeofcases]}");
print("*"*155);
print("\n\n\n");

for (int j=0;j<statesvalues.length;j++)
{
    for (int k=j+1; k<statesvalues.length;k++)
    {
        if (statesvalues[k][caseskeys[typeofcases]]<statesvalues[j][
caseskeys[typeofcases]])
        {

            var temp=statesvalues[j];
            statesvalues[j]=statesvalues[k];
            statesvalues[k]=temp;

            String temp2=stateskeys[j];
            stateskeys[j]=stateskeys[k];
            stateskeys[k]=temp2;
        }
    }
}
print("#"*155);
print("\t\t\t\t\tAfter sorting according to the ${caseskeys[
typeofcases]} cases in descending order.");
print("#"*155);

```

```

stdout.write(" | \t States \t |");

for(int r=0;r<caseskeys.length;r++)
{
    stdout.write(" \t ${caseskeys[r]}");
    if ([caseskeys[r]].length<maxlength)
    {

        int temp4=caseskeys[r].length;
        int temp5=maxlength-temp4;

        for(int y=0;y<temp5;y++)
        {
            stdout.write(" ");
        }

        stdout.write("\t|");
    }

}

for(int i=0;i<statesvalues.length;i++)
{
    stdout.write("\n | \t ${stateskeys[i]} \t |");
    for(int s=0;s<caseskeys.length;s++)
    {
        stdout.write(" \t ${statesvalues[i][caseskeys[s]]}");
        if (statesvalues[i][caseskeys[s]].toString().length<max)
        {
            var temp9=statesvalues[i][caseskeys[s]].toString();
            var temp10=temp9.length;
            int temp3=max-temp10;

            for(int y=0;y<temp3;y++)
            {
                stdout.write(" ");
            }
        }
        stdout.write("\t|");

    }

}

print("\n"*3);
}

```

```

#####
States with number of cases
#####
| States | Confirmed | Active | Recovered | Deceased | Tested |
| Maharashtra | 6229596 | 94593 | 6000911 | 130753 | 5546565 |
| Kerala | 3187716 | 126396 | 3045310 | 15512 | 554665 |
| Karnataka | 2886702 | 26256 | 2824197 | 36226 | 56565 |
| TamilNadu | 2539277 | 26717 | 2478778 | 33782 | 554685 |
| AndhraPradesh | 1946749 | 23939 | 1909613 | 13197 | 554667565 |
| UttarPradesh | 1708005 | 1036 | 1684230 | 22739 | 553565 |

Types of cases are : [Confirmed, Active, Recovered, Deceased, Tested]
Confirmed: 0
Active: 1
Recovered: 2
Deceased: 3
Tested: 4
put variable typeofcases value to 0/1/2/3 according to what you need for sorting according to that cases.
for the time being, typeofcases=1--> Active
#####

After sorting according to the Active cases in descending order.
#####
| States | Confirmed | Active | Recovered | Deceased | Tested |
| UttarPradesh | 1708005 | 1036 | 1684230 | 22739 | 553565 |
| AndhraPradesh | 1946749 | 23939 | 1909613 | 13197 | 554667565 |
| Karnataka | 2886702 | 26256 | 2824197 | 36226 | 56565 |
| TamilNadu | 2539277 | 26717 | 2478778 | 33782 | 554685 |
| Maharashtra | 6229596 | 94593 | 6000911 | 130753 | 5546565 |
| Kerala | 3187716 | 126396 | 3045310 | 15512 | 554665 |

```

2.15.3 A Program for small case study of website makemytrip using inheritance and classes concept

```

class oneWay{
    String? source , destination , departure_date;
    int? noOfTravellers;
    oneWay({this.source , this.destination , this.departure_date , this.
        noOfTravellers});
    Map toMap() {
        return{
            'source':source ,
            'destination':destination ,
            'departure_date': departure_date ,
            'noOfTravellers':noOfTravellers
        };
    }

    @override
    String toString()
    {
        return{
            'source':source ,
            'destination':destination ,
            'departure_date': departure_date ,
            'noOfTravellers':noOfTravellers
        }.toString();
    }
    // void ShowOneWay()
    // {

```



```

//    print(" Source: ${source}\n Destination: ${destination} \n
Departure Date: ${departure_date} \n No. of Travellers: ${
noOfTravellers} ");
// }
}
class roundTrip extends oneWay{
    String? returnDate;
    roundTrip({source, destination, departure_date, noOfTravellers, this
        .returnDate}):super(source: source, destination: destination,
        departure_date: departure_date, noOfTravellers: noOfTravellers);
    @override
    String toString() {
        // TODO: implement toString
        String parentData=super.toString();
        String myData={
            'returnDate':returnDate
        }.toString();
        return parentData+myData;
    }
    // void ShowRoundTrip()
    // {
    //     ShowOneWay();
    //     print("Return Date: ${returnDate}");
    // }
}
class MultiCity extends oneWay{
    int? noOfJourney;
    MultiCity({source, destination, departure_date, noOfTravellers, this
        .noOfJourney}):super(source: source, destination: destination,
        departure_date: departure_date, noOfTravellers: noOfTravellers);
    @override
    String toString() {
        // TODO: implement toString
        String parentData=super.toString();
        String myData={
            'noOfJourney':noOfJourney
        }.toString();
        return parentData+myData;}
}
void main() {
    // oneWay(source: "Delhi", destination: "Mumbai", departure_date:
    "24 July, 2021", noOfTravellers: 3).ShowOneWay();
    print(roundTrip(source: "Delhi", destination: "Mumbai",
        departure_date: "24 July, 2021", noOfTravellers: 3, returnDate: "26
        July, 2021"));
    print(MultiCity(source: "Delhi", destination: "Mumbai",
        departure_date: "24 July, 2021", noOfTravellers: 3, noOfJourney: 1)
        );
}

```

PROBLEMS 37 OUTPUT TERMINAL DEBUG CONSOLE

```
{source: Delhi, destination: Mumbai, departure_date: 24 July, 2021, noOfTravellers: 3}{returnDate: 26 July, 2021}  
{source: Delhi, destination: Mumbai, departure_date: 24 July, 2021, noOfTravellers: 3}{noOfJourney: 1}  
Exited
```

Chapter 3

Flutter

3.1 Flutter Install

3.1.1 System Requirements in Linux

To install and run Flutter, your development environment must meet these minimum requirements:

- Operating Systems: Linux (64-bit)
- Disk Space: 600 MB (does not include disk space for IDE/tools).
- Tools: Flutter depends on these command-line tools being available in your environment.
- Shared libraries: Flutter test command depends on this library being available in your environment.

3.1.2 Get the Flutter SDK

The easiest way to install Flutter on Linux is by using snapd. For more information, see [Installing snapd](#).

Once you have snapd, you can install Flutter using the Snap Store, or at the command line:

```
sudo snap install flutter --classic
```

Note: Once the snap is installed, you can use the following command to display your Flutter SDK path:

```
flutter sdk-path
```

3.1.3 Run flutter doctor

Run the following command to see if there are any dependencies you need to install to complete the setup (for verbose output, add the -v flag):

```
flutter doctor
```

This command checks your environment and displays a report to the terminal window. The Dart SDK is bundled with Flutter; it is not necessary to install Dart separately.

3.1.4 Update your path

You can update your PATH variable for the current session at the command line, as shown in Get the Flutter SDK. You'll probably want to update this variable permanently, so you can run flutter commands in any terminal session.

3.2 Android setup

- **Install Android Studio**

1. Download and install Android Studio.
2. Start Android Studio, and go through the 'Android Studio Setup Wizard'. This installs the latest Android SDK, Android SDK Command-line Tools, and Android SDK Build-Tools, which are required by Flutter when developing for Android.
3. Run flutter doctor to confirm that Flutter has located your installation of Android Studio. If Flutter cannot locate it, run flutter config --android-studio-dir <directory> to set the directory that Android Studio is installed to.

- **Set up your Android device**

To prepare to run and test your Flutter app on an Android device, you need an Android device running Android 4.1 (API level 16) or higher.

Enable Developer options and USB debugging on your device. Detailed instructions are available in the Android documentation.

- **Agree to Android Licenses**

Before you can use Flutter, you must agree to the licenses of the Android SDK platform. This step should be done after you have installed the tools listed above.

1. Make sure that you have a version of Java 8 installed and that your

```
JAVA_HOME
```

environment variable is set to the JDK's folder.

Android Studio versions 2.2 and higher come with a JDK, so this should already be done.

2. Open an elevated console window and run the following command to begin signing licenses.

```
flutter doctor --android-licenses
```

3. Review the terms of each license carefully before agreeing to them.

4. Once you are done agreeing with licenses, run flutter doctor again to confirm that you are ready to use Flutter.

3.3 Create the app

1. Open the IDE and select Create New Flutter Project.
2. Select Flutter Application as the project type. Then click Next.
3. Verify the Flutter SDK path specifies the SDK's location (select Install SDK if the text field is blank).
4. Enter a project name (for example, myapp). Then click Next. Click Finish.
5. Wait for Android Studio to install the SDK and create the project.

3.4 Write your first Flutter app

Firstly, Create a simple, templated Flutter app.

Weâll mostly edit lib/main.dart, where the Dart code lives.

Replace the contents of lib/main.dart according to the need.

Run the app in the way your IDE describes. You should see either Android, iOS, or web output, depending on your device.

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

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

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        // This is the theme of your application.
        //
        // Try running your application with "flutter run". You'll see
        the
        // application has a blue toolbar. Then, without quitting the
        app, try
        // changing the primarySwatch below to Colors.green and then
        invoke
        // "hot reload" (press "r" in the console where you ran "
        flutter run",
        // or simply save your changes to "hot reload" in a Flutter
        IDE).
        // Notice that the counter didn't reset back to zero; the
        application
        // is not restarted.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key? key, required this.title}) : super(key: key);

  // This widget is the home page of your application. It is stateful,
  meaning
  // that it has a State object (defined below) that contains fields
  that affect
  // how it looks.
```

```

// This class is the configuration for the state. It holds the
// values (in this
// case the title) provided by the parent (in this case the App
// widget) and
// used by the build method of the State. Fields in a Widget
// subclass are
// always marked "final".

final String title;

@override
_MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework that
      // something has
      // changed in this State, which causes it to rerun the build
      // method below
      // so that the display can reflect the updated values. If we
      // changed
      // _counter without calling setState(), then the build method
      // would not be
      // called again, and so nothing would appear to happen.
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called, for
    // instance as done
    // by the _incrementCounter method above.
    //
    // The Flutter framework has been optimized to make rerunning
    // build methods
    // fast, so that you can just rebuild anything that needs updating
    // rather
    // than having to individually change instances of widgets.
    return Scaffold(
      appBar: AppBar(
        // Here we take the value from the MyHomePage object that was
        // created by
        // the App.build method, and use it to set our appBar title.
        title: Text(widget.title),
      ),
      body: Center(
        // Center is a layout widget. It takes a single child and
        // positions it
        // in the middle of the parent.

```

```

        child: Column(
          // Column is also a layout widget. It takes a list of
          children and
          // arranges them vertically. By default, it sizes itself to
          fit its
          // children horizontally, and tries to be as tall as its
          parent.
          //
          // Invoke "debug painting" (press "p" in the console, choose
          the
          // "Toggle Debug Paint" action from the Flutter Inspector in
          Android
          // Studio, or the "Toggle Debug Paint" command in Visual
          Studio Code)
          // to see the wireframe for each widget.
          //
          // Column has various properties to control how it sizes
          itself and
          // how it positions its children. Here we use
          mainAxisAlignment to
          // center the children vertically; the main axis here is the
          vertical
          // axis because Columns are vertical (the cross axis would
          be
          // horizontal).
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // This trailing comma makes auto-formatting nicer for build
      methods.
    );
  }
}

```

You have pushed the button this many times:

0



Chapter 4

Work Done

4.1 Experimental setup

4.1.1

Chapter 5

Future Work

5.1

5.1.1

Chapter 6

Conclusion

References

[1] Write the reference here