Lab Manual 1-10

BATCH-B1
Group no: 4
CE86-Kavya Atulkumar Shah
CE105-Dhruvil Dhamsania

Chapter 2: Expressions, Variables & Constants

Single Line Comment:

- **// This is a single-line comment**
- **Comments are ignored by the compiler and are useful for documenting your code.**

Multi-Line Comment:

- **/* This is a multi-line comment. It spans multiple lines. */**

Nested Comments:

- **/* This is a comment. /* Inside is another comment. */ Back to the first. */**

Documentation Comments:

- **/// This is a single-line documentation comment.**
- **/** This is a block documentation comment. It can span multiple lines. */**

Print Statement:

- **print('Hello, Dart!');**
- **Outputs text to the debug console.**

Statements:

- **A command you tell the computer to do. Ends with a semicolon.**
- **Example: print('Hello, Dart!');**

Expressions:

- **A value or something that can be calculated as a value.**
- **Examples: 42, 3 + 2, 'Hello, Dart!', x**

Simple Operations:

- **Add: +**
- **Subtract: -**
- **Multiply: ***
- **Divide: /**

Examples:

- **2 + 6**
- **10 - 2**
- **2 * 4**
- **24 / 3**

Truncating Division:

- **~/**
- **Example: 22 ~/ 7 (result is 3)**

Modulo Operation:

- **%**
- **Example: 28 % 10 (result is 8)**

Order of Operations:

- **Example: 350 / 5 + 2 (division first, then addition)**
- **Use parentheses to change the order: 350 / (5 + 2) (addition first, then division)**

Importing Math Library:

- **import 'dart:math';**

Common Functions:

- **sin(45 * pi / 180) - Computes the sine of 45 degrees.**
- **cos(135 * pi / 180) - Computes the cosine of 135 degrees.**
- **sqrt(2) - Computes the square root of 2.**
- **max(5, 10) - Computes the maximum of two numbers.**
- **min(-5, -10) - Computes the minimum of two numbers.**

Variable Declaration:

- **int number = 10; - Declares an integer variable.**
- **double apple = 3.14159; - Declares a double (decimal) variable.**

Changing Variable Value:

- **number = 15; - Changes the value of the variable.**

Type Safety:

- **Dart enforces type safety, meaning a variable's type cannot be changed once it is set.**
- **Example: int myInteger = 10; myInteger = 3.14159; (This will cause an error)**

Type Inference:

- **var someNumber = 10; - Dart infers the type based on the assigned value.**

Compile-Time Constants:

- **const myConstant = 10; - Declares a compile-time constant.**

Runtime Constants:

- **final hoursSinceMidnight = DateTime.now().hour; - Declares a runtime constant.**

Valid Identifiers:

- **Must start with a letter or underscore, can contain letters, digits, and underscores.**
- **Examples: firstName, num1, $result**

Invalid Identifiers:

- **Cannot start with a digit or contain spaces or special characters (except underscore and dollar sign).**
- **Examples: 1number, first name, first-name.**

Common Keywords:

- **abstract, continue, false, new, this, if, else, while, for, return, void, var, final, const**

Increment and Decrement:

- **counter += 1; or counter++; - Increments the counter by 1.**
- **counter -= 1; or counter--; - Decrements the counter by 1.**

Chapter 3: Types & Operations

Data Types in Dart

- **int: Used for integer values.**
- **double: Used for floating-point numbers.**
- **num: Can hold either int or double values.**
- **dynamic: Can hold any type of value.**
- **String: Used for text.**

Type Inference

Dart can automatically determine the type of a variable based on its assigned value.

Annotating Variables Explicitly

You can specify the type of a variable explicitly when declaring it.

Creating Constant Variables

Using `const` or `final` makes variables immutable.

Letting the Compiler Infer the Type

Dart can infer the type of a variable based on its value.

Checking the Inferred Type in VS Code

Hovering over a variable in VS Code shows its inferred type.

Checking the Type at Runtime

You can check the type of a variable at runtime using the `is` keyword.

Type Conversion

Converting data from one type to another requires explicit conversion methods.

Operators with Mixed Types

Dart handles operators with mixed types by promoting the result to a type that preserves precision.

Ensuring a Certain Type

You can ensure a variable remains a specific type using type annotations.

Casting Down

Casting allows you to convert a variable to a more specific subtype if needed.

Strings

Working with Strings

Strings in Dart are used for representing text.

Single-Quotes vs. Double-Quotes

Dart allows both single and double quotes for defining strings.

Concatenation

Strings can be combined using the `+` operator.

String Buffer

For efficient string manipulations, use a StringBuffer.

Interpolation

String interpolation allows embedding expressions inside strings using `$`.

Multi-Line Strings

Multi-line strings in Dart are enclosed in triple quotes.

Object and Dynamic Types

Dynamic Typing

Dart supports dynamic typing with the `dynamic` keyword.

Chapter 4: Control Flow

Control Flow

Control flow refers to the order in which statements are executed in a program based on certain conditions.

Boolean Values

In programming, a Boolean value represents either true or false. It's used to indicate the result of a comparison or a logical operation.

Boolean Operators

- **Comparison Operators: These operators compare two values and return a Boolean result:**
    - **Equality: Checks if two values are equal (==).**
    - **Inequality: Checks if two values are not equal (!=).**
    - **Relational Operators: Compare values based on magnitude (>, <, >=, <=).**
- **Logical Operators: These operators combine multiple Boolean expressions:**
    - **AND (&&): Returns true only if both operands are true.**
    - **OR (||): Returns true if at least one operand is true.**
    - **NOT (!): Inverts the Boolean value (true becomes false and vice versa).**

Conditional Statements

- **if Statement: Executes a block of code if a specified condition is true.**
- **else Statement: Executes a block of code if the preceding if condition is false.**
- **else if Statement: Allows for multiple conditions to be checked sequentially if the preceding conditions are false.**

Examples:

- **Boolean Values:**
  - **isSunny = true**
  - **isRaining = false**
- **Comparison:**
  - **1 == 2 evaluates to false**
  - **1 != 2 evaluates to true**
- **Logical Operators:**
  - **true && false evaluates to false**
  - **true || false evaluates to true**
  - **!false evaluates to true**
- **Conditional Statements:**

```
if (isSunny) {
        print("It's sunny today!");
}
else {
        print("It's not sunny today.");
 }
```

Chapter 5: Loops

While Loops

- **Definition: Executes a block of code repeatedly as long as a specified condition is true.**
- **Example Explanation: Continuously adds 4 to a variable sum until it's no longer less than 10.**

Do-While Loops

- **Definition: Similar to while loop but ensures the block of code executes at least once, then checks the condition.**
- **Example Explanation: Adds 4 to a variable sum repeatedly until sum is greater than 10.**

For Loops

- **Definition: Executes a block of code a specific number of times.**
- **Example Explanation: Prints numbers from 1 to 5 by iterating an iterator variable i.**

For-In Loops

- **Definition: Simplifies iterating over elements of a collection without managing an index.**
- **Example Explanation: Iterates over a list of numbers and performs an operation on each element.**

ForEach Loops

- **Definition: Method available on collections that iterates over elements with a concise syntax.**
- **Example Explanation: Iterates over a list of numbers using forEach and performs an operation on each element.**

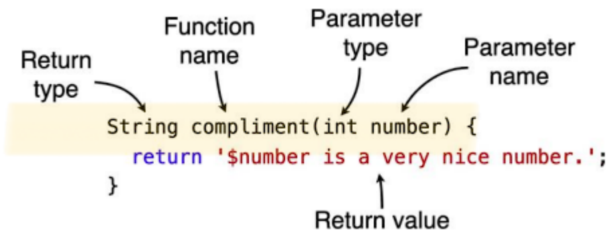
Chapter 6: Characters and Strings

1. **Character or char type: In Dart, there's no distinct char type for single characters. Even single characters are treated as strings.**
2. **UTF-16: A character encoding scheme used in Dart and many other languages to represent Unicode code points using 16-bit numbers. It has limitations for characters beyond the Basic Multilingual Plane (BMP).**
3. **Unicode code points: Unique numbers assigned to each character and symbol in the Unicode standard. They are used to identify and represent characters.**
4. **codeUnits: A Dart string property that returns a list of UTF-16 code units representing each character's numeric value.**
5. **runes: Another Dart string property that returns a collection of Unicode code points instead of UTF-16 code units, allowing access to characters beyond the BMP.**
6. **Grapheme clusters: Sequences of one or more Unicode characters that together form what users perceive as a single character. Important for handling complex characters like emojis and diacritics.**
7. **Zalgo text: A form of text where combining characters are used to modify the appearance of other characters, creating a visually distorted output.**
8. **characters package: An add-on package in Dart that enhances the String type to support operations specifically on grapheme clusters. It's useful for accurate text handling, especially with user input or network data.**

Chapter 5:Functions

Function is like machines; they take something you provide to them (the input), and produce something different (the output).

Input
Function
Output



```
String compliment(int number) {
  return '$number is a very nice number.';
}
```

In Dart, a function consists of a return type, a name, a parameter list in parentheses and a body enclosed in braces.

In Dart, parameters in functions can be made optional by using either named parameters or positional parameters.

Named Parameters:

Named parameters are enclosed in curly braces {} in the function signature. They can have default values. When invoking the function, you can specify these named parameters in any order.

```
void printUserInfo({String name = 'Guest', int age = 18}) {
  print('Name: $name, Age: $age');
}

void main() {
  printUserInfo();

  printUserInfo(name: 'Sumit');

  printUserInfo(name: 'Nana’, age: 19);
}
```

Positional Parameters

Positional parameters are enclosed in square brackets [ ] in the function signature. These parameters can also have default values. When invoking the function, the parameters must be specified in the order they are defined.

```
void printUserInfo([String name = 'Guest', int age = 18]) {
  print('Name: $name, Age: $age');
}

void main() {
  printUserInfo();
  printUserInfo('Sumit');
  printUserInfo('Nana', 19);
}
```

- In Dart, we can omit explicit return types and parameter types in function definitions. When omitted, Dart infers these types from the context or uses dynamic typing.

Omitting Return Type

The return type of a function can be omitted, and Dart will infer it from the return statement within the function.

```
greet(name) {
  return 'Hello, $name!';
}

void main() {
  var greeting = greet('Sumit');
  print(greeting);
}
```

Omitting Parameter Types

Parameter types can be omitted in a function definition. Dart will infer these types from the context or treat them as dynamic.

```
void printUserInfo(name, age) {
  print('Name: $name');
  print('Age: $age');
}

void main() {
  printUserInfo('Sumit', 19);
}
```

Omitting Both Return Type and Parameter Types

Both the return type and parameter types can be omitted in a function.

```
displayInfo(name, age, city) {
  print('Name: $name');
  print('Age: $age');
  print('City: $city');
}
```

```
void main() {
  displayInfo('Sumit', 19, 'Surat');
}
```

Anonymous Functions

In Dart, anonymous functions are functions that do not have a name. They are useful for passing as arguments to other functions or for defining short, inline functions. Anonymous functions can capture variables from their surrounding scope.

Anonymous functions in Dart can be defined using two different syntaxes:

1. **Regular function syntax**: () {}
2. **Arrow function syntax**: =>.

```
var multiply = (int a, int b) => a * b;

void main() {
  var result = multiply(3, 4);
  print(result);
}
```

Closure and Scope

Scope

Scope refers to the visibility of variables within different parts of our code. Dart has three main types of scope:

1. **Global Scope**: Variables declared outside of any function or class are in the global scope and can be accessed from anywhere in the code.
2. **Local Scope**: Variables declared within a function or block (like within a for-loop) are in the local scope and can only be accessed within that function or block.
3. **Class Scope**: Variables declared within a class (but outside of methods) are in the class scope and can be accessed by all methods within the class.

Closures

A closure is a function object that has access to variables in its lexical scope, even when the function is used outside of its original scope. In other words, closures "remember" the environment in which they were created.

Chapter 6: Classes

Classes in Dart serve as blueprints for creating objects. If a class is a blueprint, then an object is the actual structure built from that blueprint. For instance, the String class defines its data as a sequence of UTF-16 code units, while a String object represents a specific instance like "Hello!!"

In Dart, every value, including basic types like int ,double and bool is an object derived from a class. This approach contrasts with languages like Java, where basic types are treated as primitives.

Classes are central to object-oriented programming, as they encapsulate both data and functions within a single entity. Within a class:

- **Methods** are functions that operate on the class's data.
- **Constructors** are special methods used to create and initialize objects from the class.

To create your own types, start by defining a simple User class with id and name properties:

Code:

```
class User {
  int id = 0;
  String name = '';
}
```

This class has two properties: id (an int with a default value of 0) and name (a String with a default value of an empty string).

Creating an Object from a Class

To create an instance of your User class, use the following code inside the main function:

Code:

```
void main() {
  final user = User();
}
```

This creates an instance of the User class and stores it in the user variable.

Assigning Values to Properties

You can assign values to the properties of your User object using dot notation:

Code:

```
void main() {
  final user = User();
  user.name = 'Ray';
  user.id = 42;
}
```

Adding Methods

You can add your own methods to a class. For example, to serialize a User object to JSON, add the following method:

Code:

```
class User {
  int id = 0;
  String name = "";

  String toJson() {
    return '{"id": $id, "name": "$name"}';
  }

  @override
  String toString() {
    return 'User(id: $id, name: $name)';
  }
}

void main() {
  final user = User();
  user.name =”Kavya”;
  user.id = 86;
  print(user.toJson());  // Output: {"id": 86, "name": "Kavya"}
}
```

Default Constructor

If no constructor is provided, Dart supplies a default constructor with no parameters.

Example:

Code:

```
class Address {
  var value = '';
}
```

This is equivalent to:

Code:

```
class Address {
  Address();
  var value = '';
}
```

Custom Constructors

You can define a constructor to initialize properties at the time of object creation.

Long-Form Constructor Example:

Code:

```
class User {
  int id;
  String name;

  User(int id, String name) {
    this.id = id;
    this.name = name;
  }
}
```

Named Constructors

Named constructors allow you to create multiple constructors with different names for the same class.

Example:

Code:

```
class User {
  int id;
  String name;

  User(this.id, this.name);

  User.anonymous() {
    id = 0;
    name = 'anonymous';
  }
}
```

Forwarding Constructors

You can call one constructor from another using this keyword.

Example:

Code:

```
class User {
  int id;
  String name;

  User(this.id, this.name);

  User.anonymous() : this(0, 'anonymous');
}
```

Optional and Named Parameters

Dart allows you to define optional and named parameters for constructors.

Optional Parameters Example:

Code:

```
class User {
  int id;
  String name;

  User([this.id = 0, this.name = 'anonymous']);
}
```

Named Parameters Example:

Code:

```
class User {
  int id;
  String name;

  User({this.id = 0, this.name = 'anonymous'});
}
```

Private Variables

In Dart, private variables are prefixed with an underscore _.

Example:
Code:

```
class User {
  int _id;
  String _name;

  User({int id = 0, String name = 'anonymous'})
```

```
    : _id = id,
      _name = name;
}
```

Initializer Lists

Initializer lists are used to initialize fields before the constructor body runs.

Example:

Code:

```
class User {
  int _id;
  String _name;

  User({int id = 0, String name = 'anonymous'})
    : _id = id,
      _name = name;
}
```

Now, Putting It All Together

Here’s a complete example that uses many of these concepts:

Code:

```
class User {
  int _id;
  String _name;

  User({int id = 0, String name = 'anonymous'})
    : _id = id,
      _name = name;

  User.anonymous() : this();

  @override
  String toString() => 'User(id: $_id, name: $_name)';
}

void main() {
  final user = User(id: 42, name: 'Ray');
  print(user); // Output: User(id: 42, name: Ray)

  final anonymousUser = User.anonymous();
  print(anonymousUser); // Output: User(id: 0, name: anonymous)
}
```

In this example, we have a User class with a main constructor that uses named parameters, a named constructor for creating an anonymous user, private variables to prevent unauthorized access, and an initializer list to set values before the constructor body executes.

Understanding Privacy in Dart

In Dart, prefixing a variable or method name with an underscore (_) makes it "library private" rather than "class private." This means that the member can be accessed by any code within the same library (file), but not from outside the library.

```
class User {
  String _name;

  User(this._name);
}

void main() {
  var user = User('John');
  print(user._name);
}
```

Making Properties Truly Private

To make properties truly private, you need to separate the class definition into a different file. Here's how you can do it:

Create a separate file for the User class:

Code:

```
// user.dart
class User {
  String _name;

  User(this._name);

  String get name => _name; // Getter for _name
}
```

Use the User class in the main file:

Code:

```
import 'user.dart';

void main() {
  var user = User('John');
  // user._name = 'Hacker'; // This will now cause an error
  print(user.name); // This will work
}
```

Error Checking with Initializer Lists

You can add error checking in Dart constructors using initializer lists. This is useful for ensuring that the parameters meet certain conditions before the object is created.

Code:

```
class User {
  final int id;
  final String name;

  User({this.id = 0, this.name = 'anonymous'})
      : assert(id >= 0, 'ID must be non-negative'),
        assert(name.isNotEmpty, 'Name cannot be empty');
}

void main() {
  var user = User(id: -1, name: 'John');
}
```

Making Properties Immutable

To make properties immutable, you can use the final keyword, which ensures the properties can only be set once.

code

```
class User {
  final int id;
  final String name;

  User({required this.id, required this.name});
}

void main() {
  var user = User(id: 1, name: 'John');
  // user.id = 2; // This will cause an error
}
```

Using const Constructors

If all properties of a class are final, you can use a const constructor to create compile-time constants.

Code:

```
class User {
  final int id;
  final String name;

  const User({this.id = 0, this.name = 'anonymous'});

  const User.anonymous() : this();
}

void main() {
  const user = User(id: 1, name: 'John');
  const anonymousUser = User.anonymous();
}
```

Factory Constructors

Factory constructors provide more flexibility by allowing the constructor to return an instance of a class or its subclass, rather than always creating a new instance.

Code:

```
class User {
  final int id;
  final String name;

  const User({required this.id, required this.name});

  factory User.fromJson(Map<String, Object> json) {
    final userId = json['id'] as int;
    final userName = json['name'] as String;
    return User(id: userId, name: userName);
  }
}

void main() {
  final map = {'id': 10, 'name': 'Manda'};
  final user = User.fromJson(map);
}
```

Getters and Setters

Getters and setters in Dart allow controlled access to class properties.

Code:

```
class Email {
  String _address = '';

  String get address => _address;
  set address(String newAddress) => _address = newAddress;
}

void main() {
  var email = Email();
  email.address = 'ray@example.com';
  print(email.address);
}
```

Static Members

Static members belong to the class rather than any instance.

```
class MyClass {
  static int myProperty = 0;

  static void myMethod() {
    print('Hello, Dart!');
  }
}

void main() {
  print(MyClass.myProperty);
  MyClass.myMethod();
}
```

Singleton Pattern

A singleton class ensures only one instance is created.

Code:

```
class MySingleton {
  MySingleton._();
  static final MySingleton instance = MySingleton._();
}

void main() {
  var singleton = MySingleton.instance;
}
```

Summary

To achieve true encapsulation and privacy in Dart:

1. **Library Privacy**: Use an underscore (_) to make properties library-private, meaning they are only accessible within the same file. For true encapsulation, properties should be placed in a different library file.
2. **Immutability**: Use final and const to ensure immutability:
   - Final variables can be set once and are immutable after initialization.
   - Const variables are compile-time constants and are also immutable.

3. **Factory Constructors**: Provide flexibility in creating instances, allowing for controlled object creation.
4. **Getters and Setters**: Allow controlled access to private properties, providing encapsulation and validation.
5. **Static Members**: Belong to the class itself rather than any instance, allowing shared data and methods across all instances of the class.
6. **Singleton Pattern**: Ensures that only one instance of a class exists throughout the application, providing a global point of access.

This approach combines privacy, immutability, controlled access, and single-instance management to effectively manage data and functionality in Dart.

Chapter 8: Collections

I. Lists

Creating a list

var desserts = ['cookies', 'cupcakes', 'donuts','pie'];

List<String> snacks = [];

var snacks = <String>[];

The angle brackets < > here are the notation for generic types in Dart. A generic list means you can have a list of anything; you just put the type you want inside the angle brackets. In this case, you have a list of strings, but you could replace String with any other type. For example, List<int> would make a list of integers,

List<bool> would make a list of Booleans, and List<Grievance> would make a list of grievances — but you'd have to define that type yourself since Dart doesn't come with any by default.

Accessing elements final

secondELement = desserts[1];

print (secondElement); //

cupcakes

Use **i**ndexOf() method to find index of list element final index =
desserts.indexOf('pie'); // 3

final value = desserts[index]; // pie **Assigning**

values to list elements desserts[1] = 'cake';

print(desserts); // [cookies, cake, donuts, pie]

Adding elements to a list desserts**.add**('brownies'); print(desserts); // [cookies,
cake, donuts, pie, brownies] **Removing**
elements from a list

deserts**.remove**('cake');

print(desserts); // [cookies, donuts, pie, brownies]

II. <u>Mutable and Immutable lists</u>

Immutable list

final desserts = [cookies', 'cupcakes', 'donuts', 'pie']; desserts=['donuts', 'pie'] //not
allowed

Deeply Immutable list const desserts = ['cookies',

'cupcakes', 'donuts', 'pie'];

desserts.add('brownie'); // not allowed desserts.remove('pie'); // not
allowed desserts[0] = 'fudge'; // not
allowed

If you want an immutable list but you won't know the element values until runtime, then

you can create one with the List.unmodifiable named constructor: **List.unmodifiable**

final modifiableList = [DateTime.now(), DateTime.now()]; final
unmodifiableList = **List.unmodifiable**(modifiableList);

DateTime.now()returns the date and time when it's called. You're obviously not going to know that untilruntime, so this prevents the list from taking const. Passing that list into List.unmodifiable, however, makes the new list immutable.

List Properties

const drinks = ['water', 'milk', 'juice', 'soda'];

Accessing first and last elements

drinks**.**first // water

drinks**.**last // soda

Looping over the elements of list

for (var drink in drinks) { print(drink);

}

Run any of the loops above and you'll get the same
result. cookies cupcakes donuts pie

Flutter UI code is composed of classes called widgets.

Three common Flutter widgets are rows, columns and stacks, which all store their

children as List collections. **Spread operator**

const pastries = ['cookies', 'croissants']; const candy =
['Junior Mints', 'Twizzlers', 'M&Ms'];

const desserts = ['donuts', ...pastries, ...candy];

print(desserts); //[donuts, cookies, croissants, Junior Mints, Twizzlers, M&Ms] **Null spread operator**

List<String>? coffees; final hotDrinks
= ['milk tea', ...?coffees];

Here coffees has not been initialized and therefore is null. By using the **...?** operator, you avoid an error that would come by trying to add a null list. The list hotDrinks will only include milk tea.

Collection if const peanutAllergy =
true; const

```
candy = [
        'Junior Mints', 'Twizzlers', if
        (!peanutAllergy)
            'Reeses',
 ];
print(candy); //[Junior Mints, Twizzlers]
```

Run that and you'll see that the false condition for the collection if prevented Reeses from being included in the list .

III. <u>Sets</u>

Creating a set

final Set<int> someSet = {};

final someSet = <int>{};

final anotherSet = {1, 2, 3, 1};

Checking the contents

```
print(anotherSet.contains(1)); // true
print(anotherSet.contains(99)); // false
```

Adding single elements

```
final someSet = <int>{};
someSet.add(42);
someSet.add(2112);
someSet.add(42);
print(someSet);   //{42, 2112}
```

Removing elements

```
someSet.remove(2112);   //{42}
```

Adding multiple elements

```
someSet.addAll([1, 2, 3, 4]);  //{42, 1, 2, 3, 4}
```

Intersections and Unions

```
final setA = {8, 2, 3, 1, 4};
final setB = {1, 6, 5, 4};
final intersection = setA.intersection(setB); //{1, 4}
```

Unions

final union = setA.union(setB);   //{8, 2, 3, 1, 4, 6, 5}

IV. Intersections and Unions

final intersection = setA.intersection(setB); //{1, 4}

Unions final union = setA.union(setB); //{8, 2, 3,

1, 4, 6, 5}

V. Maps

Creating a Map

final Map<String, int> emptyMap = {};
final emptyMap = <String, int>{}; final
emptySomething = {};

will infer it to Map if you want set then be explicit final

inventory = {

'cakes': 20,

'pies': 14,

'donuts': 37,

'cookies': 141,

};

Printing Maps

print(emptyMap.length); //0

print(inventory); //{cakes: 20, pies: 14, donuts: 37, cookies: 141} **Unique keys**

```
final treasureMap = {
            'garbage': ['in the dumpster'],
            'glasses': ['on your head'],
            'gold': ['in the cave', 'under your mattress'],
    };
```

Accessing elements from a map

```
final numberOfCakes = inventory['cakes'];
    print(numberOfCakes) //20
```

Adding elements to a map

```
    inventory['brownies'] = 3;
```

Updating an element

```
inventory['cakes'] = 1;
```

Removing elements from a map

```
    inventory.remove('cookies');
```

Map Properties

```
inventory.isEmpty // false
      inventory.isNotEmpty // true inventory.length      // 4
```

Looping over elements of a map

```
    for (var item in inventory.keys) { print(inventory[item]);
}
```

inventory.forEach((key, value) => print('$key -> $value')); for
(final entry in inventory.entries) { print('${entry.key} >
${entry.value}');
}

VII. <u>Higher Order methods</u>

Mapping over a collection

const numbers = [1, 2, 3, 4]; final squares = numbers**.**map((number) =>
number * number); print(square); //(1, 4, 9, 16)

Mapping allows you to transform each element in a collection using the map method, which takes an anonymous function and returns a new iterable with the transformed elements.

Filtering a collection

final evens = squares**.**where((square) => square.isEven);

print(evens); //(4, 16)

The where method filters elements based on a condition provided by an
anonymous function, returning only those that meet the criteria **Using Reduce**

const amounts = [199, 299, 299, 199, 499];

final total = amounts.reduce((sum, element) => sum + element); print(total);
//1495

Combines elements into a single value.

Using Fold

const amounts = [199, 299, 299, 199, 499]; final total = amounts.fold(0,
(int sum, element) => sum + element); print(total); //1495

Similar to reduce but includes an initial value, useful for empty lists.

Sorting a list

```
final desserts = ['cookies', 'pie', 'donuts', 'brownies'];
desserts.sort();
print(desserts); //[brownies, cookies, donuts, pie]
```

Reversing a list

```
var dessertsReversed = desserts.reversed;
```

Custom sort

```
desserts.sort((d1, d2) => d1.length.compareTo(d2.length));
```

Combining higher order methods

```
const desserts = ['cake', 'pie', 'donuts', 'brownies']; final
bigTallDesserts = desserts .where(
(dessert) => dessert.length > 5)
.map((dessert) => dessert.toUpperCase());
```

Chapter 9: Advanced Classes

I. Extending classes

Inheritance allows creating a hierarchy of classes where child classes inherit properties and methods from parent classes. This helps in code reuse and creating structured class hierarchies.

Creating Your First Subclass

First, define an enumeration for grades: enum

```
Grade { A, B, C, D, F }
```

Creating Similar Classes Person Class : class Person {

```
 Person(this.givenName, this.surname);
 String givenName;
 String surname;
 String get fullName => '$givenName $surname';

 @override
 String toString() => fullName;
}
```

Student Class :

```
class Student {
 Student(this.givenName, this.surname);
 String givenName; String
surname; var grades =
<Grade>[];
 String get fullName => '$givenName $surname';

 @override
 String toString() => fullName;
}
```

Subclassing to Remove Code Duplication Remove duplication by making Student extend Person:

```
class Student extends Person {
 Student(String givenName, String surname) : super(givenName,
surname);
 var grades = <Grade>[];
}
```

Using the super Keyword

The super keyword is used to call the constructor of the parent class: class Student extends Person {

```
 Student(String givenName, String surname) : super(givenName, surname); var grades
= <Grade>[];
}
```

Using the Classes

Create instances of Person and Student and demonstrate how the inheritance works:

```
void main() {
 final jon =Person('Jon', 'Snow');
 final jane = Student('Jane', 'Snow');
          print(jon.fullName);

          // Outputs: Jon Snow print(jane.fullName);

           // Outputs: Jane Snow

 final historyGrade = Grade.B;
jane.grades.add(historyGrade);
 print(jane.grades); // Outputs: [Grade.B]
}
```

Overriding Parent Methods

To change how a student's full name is printed compared to the default way in Person, you can override the fullName getter in the Student class class Student extends Person {

```
 Student(String givenName, String surname) : super(givenName, surname); var grades =
<Grade>[];

 @override
 String get fullName => '$surname, $givenName';
}
```

Calling ‘super’ from an Overridden Method

Sometimes you might want to add to the functionality of a parent method rather than completely replace it. You can call the parent method using super.

```
class SomeParent {
void doSomeWork()
{
 print('parent working');
 }
}

 class SomeChild extends SomeParent {
 @override void doSomeWork()
 {
  super.doSomeWork();
  print('child doing some other
  work');
 } }
```

Multi-Level Hierarchy

```
class Person {

  Person(this.givenName, this.surname);
  String givenName;
  String surname;
  String get fullName => '$givenName $surname';

  @override
  String toString() => fullName;
 }

 class Student extends Person {
  Student(String givenName, String surname) : super(givenName, surname);
  var grades = <Grade>[];
 }

class SchoolBandMember extends Student {

  SchoolBandMember(String givenName, String surname)
  : super(givenName, surname); static
  const minimumPracticeTime = 2;
 }
```

Type Inference in a Mixed List

Since jane, jessie, and marty are all instances of classes that extend Student, the list's type is inferred as List<Student>:

```
final students = [jane, jessie, marty];
```

Checking an Object's Type at Runtime

```
print(jessie is Object); // true print(jessie
 is Person); // true print(jessie is
 Student); // true print(jessie is
 SchoolBandMember); // true print(jessie is!
 StudentAthlete); // true
```

II. Abstract classes

Abstract classes in object-oriented programming are a way to define a blueprint for other classes. Unlike concrete classes, you cannot create instances of abstract classes directly. Instead, abstract classes are used to specify general characteristics and behaviors that other, more specific subclasses must implement. They allow you to outline methods and properties that subclasses should provide concrete implementations for. This approach helps in organizing code and promoting reusability by focusing on the general structure rather than the specifics.

Creating Your Own Abstract Classes

Define an abstract class using the `abstract` keyword. An abstract class cannot be instantiated directly but can include abstract methods that must be implemented by subclasses.

```
abstract class Animal {

    bool isAlive = true;


    void
    eat();
    void
    move()
    ;

    @override
    String toString() { return
    "I'm a $runtimeType";
    }
   }
```

Can’t Instantiate Abstract Classes

Attempting to create an instance of an abstract class will result in an error: final animal = Animal(); // Error: Abstract classes can’t be instantiated.

Concrete Subclass

To use an abstract class, you need to create a concrete subclass that provides implementations for all abstract methods:

```
class Platypus extends Animal {}
```

Filling in the TODOs

Implement the methods in your concrete class:

```
@overri
de void
eat() {
 print('Munch munch');
}

@override void move()
 {
  print('Glide
  glide');
}
```

Add additional methods specific to the subclass:

```
void layEggs() {
 print('Plop plop');
}
```

Testing the Results

Create an instance of the concrete subclass and test its functionality: void main() {

```
Final platypus = Platypus();
 print(platypus.isAlive); // true
platypus.eat();

 // Munch munch platypus.move();

 // Glide glide platypus.layEggs();

// Plop plop print(platypus); // I'm a Platypus }
```

Treating Concrete Classes as Abstract

You can use abstract class types to refer to concrete instances, focusing on their abstract characteristics:

```
Animal platypus = Platypus();
```

```
print(platypus); // I'm a Platypus
```

In this case, Dart treats `platypus` as an `Animal`, which is useful when you only need to work with the general properties and methods defined in the abstract class.

III. Interfaces

Interfaces in Dart allow you to define the behavior you expect for all classes that implement the interface. They help in separating implementation details from the rest of your code, making it more scalable, maintainable, and testable.

Software Architecture

In software architecture, it's crucial to keep core business logic separate from infrastructure like UI, database, and network. This separation ensures that changes in infrastructure do not destabilize the stable business logic.

Communication Rules

Interfaces act as a protocol or set of rules for communication between different parts of your codebase. This allows different parts of the code to change independently as long as they adhere to the interface.

Separating Business Logic from Infrastructure

Using interfaces, the business logic can interact with different forms of storage or services without knowing the specific implementation details. This makes the application architecture more flexible.

Creating an Interface

In Dart, you can create an interface using an abstract class.

```
abstract class DataRepository {
    double? fetchTemperature(String city);
    }
```

Implementing the Interface

Use the `implements` keyword to create a concrete class that provides the implementation for the interface.

```
class FakeWebServer implements DataRepository {
 @override
 double? fetchTemperature(String city) {
  return 42.0; // Dummy implementation
 }
}
```

Using the Interface

You can use the interface in your business logic without worrying about the concrete implementation.

```
void main() {
  final DataRepository repository = FakeWebServer();
   final temperature = repository.fetchTemperature('Berlin');
 print(temperature); // Output: 42.0
}
```

Adding a Factory Constructor

A factory constructor can return a subclass, making it easier to swap implementations.

```
abstract class DataRepository {
 factory DataRepository() => FakeWebServer();

double? fetchTemperature(String city);

}
```

Interfaces and the Dart SDK

Dart heavily uses interfaces to define its API, allowing the implementation details to change without affecting developers.

Extending vs Implementing

Extending : Inherits both the interface and the implementation from the superclass.

Implementing : Only inherits the interface (method signatures and field types) but requires you to provide your own implementation.

Example of Extending

```
    class AnotherClass { int

myField = 42;

     void myMethod() =>
    print(myField);

}

    class SomeClass extends AnotherClass {}


void main() {

     final someClass = SomeClass(); print(someClass.myField); //
     Output: 42
    someClass.myMethod(); // Output: 42
    }
```

IV. Mixins

Mixins in Dart allow you to reuse methods or variables among unrelated classes. They are a powerful tool for avoiding code duplication and adhering to the DRY (Don't Repeat Yourself) principle.

Problems with Extending and Implementing

Using only inheritance or interfaces can lead to code duplication, especially when classes share some behavior but are otherwise unrelated.

```
abstract class Bird {
 void
 fly();
 void
 layEggs(
 );
}

class Robin extends Bird {
 @override void
 fly() {
 print('Swoosh
 swoosh');
 }
 @override void
layEggs() {
print('Plop plop');
 }
}
```

If you want to add another class like `Platypus` that also lays eggs but does not fly, you face challenges with code duplication and inheritance limitations.

Mixing in Code

Mixins solve the problem of code duplication by allowing you to create reusable pieces of functionality that can be "mixed in" to any class.

```
mixin EggLayer { void layEggs()
{

        print('Plop plop');
        }
       }

mixin Flyer {

void fly() {
 print('Swoosh swoosh');
 }
}

class Robin extends Bird with EggLayer, Flyer {}

class Platypus extends Animal with EggLayer {
 @overri
de void
eat() {
 print('Munch munch');
 }
 @override void
move() {
 print('Glide glide');
 }
}

void main() {

 final platypus = Platypus();
 final robin = Robin();

 platypus.layEggs(); // Output: Plop plop
robin.layEggs(); // Output: Plop plop
}
```

V. Extension Methods

Extension methods in Dart allow you to add functionality to existing classes without modifying them. You can add methods, getters, setters, and operators to classes from core libraries or external packages.

Extension Syntax

To create an extension, use the following syntax:

```
 extension on SomeClass
  { // your custom code

}
```

You can also name the extension:

```
extension YourExtensionName on ClassName {

// your custom code

        }
```

Converting to an Extension

Convert the `encode` function to an extension method.

```
extension on String { String get encoded {
final output = StringBuffer(); for (final
codePoint in runes) {
output.writeCharCode(codePoint + 1);
 }
 return output.toString();
 }
}

final secret = 'abc'.encoded; print(secret);
// Output: bcd
```

Adding a Decode Extension

Add a method to decode the encoded string.

```
extension on String {
String get encoded {
return _code(1);
 }
 String get decoded {
 return _code(-1);
 }
 String _code(int step) { final output =
StringBuffer(); for (final codePoint in
runes) { output.writeCharCode(codePoint +
step);
 }
 return output.toString();
 }
}

final original = 'I like extensions!'; final secret
= original.encoded;
final revealed = secret.decoded; print(secret); // Output:
J!mjlf!fyufotjpot" print(revealed); // Output: I like extensions!
```

int Extension Example

Create an extension to calculate the cube of an integer.

```
extension on int {

int get cubed {

      return this * this * this;
      }
     }

print(5.cubed);

 // Output: 125
```

Chapter 10: Asynchronous Programming

I. Concurrency

Concurrency is when multiple tasks take turns running on a single CPU core. A thread is a sequence of commands that a computer executes . Some programming languages support multithreading, which is running multiple threads at the same time, Dart, in particular, is a single-threaded language.

II. Parallelism

Parallelism is when multiple tasks run at the same time on multiple processors or CPU cores.

Parallelism vs. concurrency

Parallelism is when multiple tasks run at the same time on multiple processors or CPU cores. Concurrency, on the other hand, is when multiple tasks take turns running on a single CPU core.
Ex. When a restaurant has a single person alternately taking orders and clearing tables, that's concurrency. But a restaurant that has one person taking orders and a different person clearing tables, that's parallelism.

problem with parallelism

in parallel threads that have access to the same memory. One thread saves a value in memory and expects the value to be the same when the thread checks the value later. However, if a second thread modifies the value, the first thread gets confused.

III. Isolate

Dart's single thread runs in what it calls an isolate. Each isolate has its own allocated memory area, which ensures that no isolate can access any other isolate's state.

IV. Synchronous vs. asynchronous code

Synchronous code is where each instruction is executed in order, one line of code immediately following the previous one. with asynchronous code, certain tasks are rescheduled to be run in the future when the thread isn't busy.

Synchronous

```
print('One'); //One
```

print('Two'); //Two

print('Three'); {

asynchronou
print('One'); **s** void main()
//three

```
Future.delayed(Duration(seconds: 2), () {
 print('Two');
});
print('Three');
}
```

output :

```
One
T
hre
e
Tw
o
```

The event loop

The event loop has two queues: a microtask queue and an event queue. The microtask queue is mostly used internally by Dart. The event queue is for events like a user entering a keystroke or touching the screen, or data coming from a database, file, or remote server.

Synchronous tasks in the main isolate thread are always run immediately. You can’t interrupt them.

If Dart finds any long-running tasks that agree to be postponed, Dart puts them In the event queue.

When Dart is finished running the synchronous tasks, the event loop checks the microtask queue. If the microtask queue has any tasks, the event loop puts them on the main thread to execute next. The event loop keeps checking the microtask queue until it’s empty.

v. Futures

The Future type

Dart has a type called Future, which is basically a promise to give you the value you really want later. Future itself is generic it can provide any type. **States for a future**

1 - Uncompleted

2 - Completed with a value **3** - Completed with an error

Getting the result with callbacks

```
 import 'dart:async';
void main() {
          print('first');
          number().then((value){
         print("value "+value);
         }).catchError((error){
         print("error "+error);
         }).whenComplete(() => {
         print("number rechived")
          });
          print('
          last');
 }
 Future<String> number() {
 return Future.delayed(Duration(seconds: 2), () => 'five');               }
```

Output :

```
First
Last
Value five
Number rechived
```

Getting the result with async-await

```
 import 'dart:async';
void main() {
            print('first');
              Future<void> number() async { print("function start");
                            final value = Future<String>.delayed(Duration(seconds: 2), () => 'five');
              print('function end');
            };
            number();
            print('last');
}
```

Output:

```
 First
 Functi
on start
Functi
on end
last
```

VI. try-catch blocks

Try-catch blocks with async-await

```
 import 'dart:async';
void main()async{
              print('fi
              rst');
              try{
               final show=await Future<String>.delayed(Duration(seconds: 2), () =>
              'five');
               print("show "+show);
               }catch(error){ print("error
               "+error.toString());
               }finally{
               print("number rechived");
           };
           print
```

```
            ('last'
            );
  }
```

Output :

```
 first
show five
number
 rechived last
```

Catching an error

```
    import
 'dart:async';
 void main()
 async{
 print('first');
 try{
  final show=await Future<String>.delayed(Duration(seconds: 2), () => 'five');
 print("show "+show); throw Exception('hy');
  } catch(error){
 print("error
 "+error.toString());
  } finally{
 print("number
 rechived");
  };
  print('last');
 }
```

Output :

```
  first
  show five error
 Exception: hy
 number
 rechived last
```

Asynchronous network requests

we use Future delay to simulate a task that takes a long so here also for future delays we can implement an interface with a mock network request class to see how your UI will react while the app is waiting for a response.

VII. data class

The data will be in JSON format, so in order to convert that into a more usable Dart object, you'll create a special class to hold the data.

```
   class Student {
Student({ required
this.studentId, required
this.name, required
this.age,
 required this.grade,
 });

 factory Student.fromJson(Map<String, Object?> jsonMap) {
 return Student( studentId:
  jsonMap['studentId'] as int,
name: jsonMap['name'] as String, age: jsonMap['age'] as
int, grade:
jsonMap['grade'] as String,
 );
 }
 final int studentId;
final String name; final int
age;
 final String grade;

 @override
 String toString() { return
 'studentId: $studentId\n'
 'name: $name\n'
 'age: $age\n'
 'grade: $grade';
 }
}
```

```
void main() {

 Map<String, Object?> jsonData = {
 'studentId': 1,
 'name': 'Raj',
 'age': 20,
 'grade': 'A',
 };
 Student student = Student.fromJson(jsonData); print(student);
}
```

Output :

```
studentId: 1
name: Raj age:
20
gra
de:
A
```

Adding the necessary imports

The http package from the Dart lets us make a GET request to a real server The dart:convert library will give jsonDecode, a function for converting a raw JSON string to a Dart map The dart:io library has HttpException and SocketException. The final import is the http library that you just added to pubspec.yaml .

Handling errors body of the main

function with a try block:

```
try { final url = 'https://jsonplaceholder.typicode.com/todos/1';
}
```

Below the try block, add catch blocks:

```
on SocketException catch (error) { print(error);
} on HttpException catch (error) { print(error);
} on FormatException catch (error) { print(error);
}
```

These catch blocks handle specific types of errors. You use the on keyword to specify the name of the exception.

SocketException

You'll get this exception if there's no internet connection. The http.get method is the one to throw the exception.

```
SocketException: Failed host lookup: 'jsonplaceholder.typicode.com'
```

HttpException

You're throwing this exception yourself if the status code is not 200 OK.

```
HttpException: 404
```

FormatException

jsonDecode throws this exception if the JSON string from the server isn't in proper JSON format. It would be unwise to blindly trust whatever the server gives you. FormatException: Unexpected character (at character 1) abc

VI. Streams

A stream is represents multiple values that will arrive in the future. Think of a stream like a list of futures.Streams, which are of type Stream, are used extensively in Dart and Dart-based frameworks. Here are some examples:

1. Reading a large file stored locally where new data from the file comes in chunks.
2. Downloading a file from a remote server.
3. Listening for requests coming into a server.
4. Representing user events such as button clicks.
5. Relaying changes in app state to the UI.

Adding an assets file

Create a new folder named assets in the root of your project . In that folder, create a file named text.txt. Add some text to the file.

Reading as a string

```
import 'dart:io';
Future<void> main() async { final file =
        File('assets/text.txt'); final contents = await file.readAsString();
        print(contents); //will
        print the text form file
}
```

Increasing the file size

In order to get a text file large enough to stream in chunks, create a new file in the assets folder called text_long.txt. Copy the Lorem Ipsum text and paste it in text_long.txt

Reading from a stream

```
 final file = File('assets/text_long.txt');
final stream = file.openRead(); stream.listen((data)
 {
  print(data.le
  ngth);
 });
```

File takes the relative path to yourtext file as the argument. Read As String returns Future<String>, but by using await you'll receive the string itself when it's ready

Using an asynchronous for loop

Just as you can use callbacks or async-await to get the value of a future, there are also two ways to get the values of a stream

```
Future<void> main() async { final file =
      File('assets/text_long.txt'); final stream
      = file.openRead(); await for
      (var data in stream) {
      print(data.length);
      }
 }
```

VII. Error handling

Using a callback One way to handle errors is to use the onError callback final file =
File('assets/text_long.txt'); final

```
stream = file.openRead(); stream.listen((data) {
print(data.length);
        },
        onError:(error) {
         print(error);
        },
        onDone:() {
         print('All finished');
        },
);
```

Using
try-catch

```
try { final
file =
File('assets/text_long.txt'); final stream = file.openRead();
await for (var data in
stream) { print(data.length);
          }
         } on Exception catch (error) {
         print(error); } finally {
          print('All finished');
 }
```

Cancelling a stream

```
import 'dart:async';
 import 'dart:io';
Future<void> main() async {
 final file = File('assets/text_long.txt');
final stream = file.openRead();
```

```
 StreamSubscription<List<int>>? subscription; subscription =
 stream.listen(
 (data) {
print(data.length);
 subscription?.cancel();
 },
 cancelOnError: true,
onDone: () {
 print('All finished');
 }
,
)
;
}
```

Calling listen returns a StreamSubscription, which is part of the dart:async library. Keeping a reference to that in the subscription variable allows you to cancel the subscription whenever you want. In this case, you cancel it afterthe first data event.

Viewing the bytes

```
import 'dart:async';
import 'dart:io';

void main() {
 final file = File('assets/text.txt');
final stream = file.openRead(); stream.listen(
 (data) {
  print(data);
 }
,
)
;
}
```

Decoding the bytes

```
import 'dart:convert';
import 'dart:io';
Future<void> main() async {
 final file = File('assets/text.txt');
final stream = file.openRead();
 await for (var data in stream.transform(utf8.decoder)) {
print(data);
 }
}
```

VIII. Isolates

App stopping synchronous code

```
 String playHideAndSeekTheLongVersion() { var
counting = 0;
 for (var i = 1; i <= 10000000000; i++) {
counting = i;
 }
return '$counting! Ready or not, here I come!';          }
```

You'll notice a significant wait until the counting finishes.

App stopping asynchronous code

```
void main() {
 Future<String> playHideAndSeekTheLongVersion() async { var
counting = 0; await Future(() {
 for (var i = 1; i <= 10000000000; i++) {
counting = i;
 }
});
 return '$counting! Ready or not, here I come!';
 }
}
```

Spawning an isolate

The term for creating an isolate in Dart is called spawning .Since isolates don't share any memory with each other, they can only communicate by sending messages. When you spawn a new isolate you give it a message communication object called a send port. The new isolate uses the send port to send messages back to a, which is listening on the main isolate.

In this example, the communication is only one way, although it's also possible to set up two-way communication between isolates.

Using a send port to return results

```
import 'dart:isolate';
void playHideAndSeekTheLongVersion(SendPort sendPort)
    { var counting = 0; for (var i = 1; i <= 1000000000; i++) {
    counting = i;
}
sendPort.send('$counting! Ready or not, here Icome!');
}
```